JAGUAR

Technical Reference Manual

Tom & Jerry

28 February, 2001

Revision 8

by Martin Brennan, Tim Dunn
and John Mathieson

Table of Contents

Introduction

This document is the Jaguar Technical Reference Manual - it is a definitive reference work for the programmer's view of the Jaguar ASICs. It is neither a hardware reference work nor a guide to a particular implementation of the Jaguar design.

This document covers the Tom and Jerry chip set. Users of the earlier prototype Jaguar silicon should consult the Appendix on the differences and enhancements. This document does not describe the prototype silicon, Revision 4 is the definitive work.

What is Jaguar?

Jaguar is a custom chip set primarily intended to be the heart of a very high-performance games / leisure computer. It may also be used as a graphics accelerator in more complex systems, and applied to work-station and business uses.

As well as a general purpose CPU, Jaguar contains four processing units. These are:

- **Object Processor**

  The Object Processor is responsible for generating the display. For each display line it processes a set of commands - the object list - and generates the display for that line in an internal line buffer.

  Objects may be bit maps in a range of display resolutions, they may be scaled, conditional actions may be performed within the object list, and interrupts to the Graphics Processor may be generated.

- **Graphics Processor**

  The Graphics Processor is a very fast micro-processor which is optimised for performing graphics generation. It has its own local RAM, and a powerful ALU which includes fast multiply and divide operations.

- **Blitter**

  The Blitter is closely coupled to the GPU, and is able to rapidly move and fill graphical objects in memory. It includes hardware support for Z-buffering and shading at very high speed.

- **Digital Sound Processor**

  The Digital Sound Processor is similar to the Graphics Processor, but is intended primarily for synthesizing sound, and for playing back sampled sound. It may also be used for general processing tasks.

Jaguar provides these blocks with a 64-bit data path to external memory devices, and is capable of a very high data transfer rate into external dynamic RAM.

How is Jaguar used?

Jaguar contains two custom chips, code-named Tom and Jerry

For graphics, Tom contains the Object Processor, the Blitter and the Graphics Processor For sound, Jerry holds the Digital Sound Processor In addition to these, there is an external CPU, currently a 68000 When animating graphics there are therefore four processing elements, all of which have got specific roles to play

The CPU is used as a manager It deals with communications with the outside world, and manages the system for the other processors It is the highest level in the control flow of a Jaguar program, and has complete control of the system

The Object Processor is at the other end of the chain for generating graphics It reads an ***object list***, and on the basis of the commands there assembles each display line of the video picture Objects are usually areas of pixels, and these may overlap and may be easily moved from frame to frame The order in which they are processed in the object list determines how they overlap Objects can also modify what is already in the display line being assembled, and can scale bit-maps They may contain transparent pixels

The Object Processor performs all the functions of a traditional ***sprite engine***, while also offering all the flexibility of a pixel-map based system It is capable of a range of animation effects, and is a powerful graphics tool in its own right

The Graphics Processor and Blitter provide a tightly coupled pair of processors for performing a much wider range of animation effects A design goal of this system was to provide a fast throughput when rendering 3D polygons The Graphics Processor therefore has a fast instruction throughput, and a powerful ALU with a parallel multiplier, a barrel-shifter, and a divide unit, in addition to the normal arithmetic functions

The Graphics Processor has four kilobytes of fast internal RAM, which is used for local program and data space This allows it to execute programs in parallel with the other processing units

The Blitter is capable of performing a range of blitting operation 64 bits at a time, allowing fast block move and fill operations, and it can generate strips of pixels for Gouraud shaded Z-buffered polygons 64 bits at a time It is also capable of rotating bit-maps, line-drawing, character-painting, and a range of other effects

The graphics processor and the Blitter will usually act together preparing bit-maps in memory, which are then displayed by the Object Processor

The DSP has eight kilobytes of fast internal RAM, and is tightly coupled to audio DACs, and has its own timers with related interrupt controller

Jaguar Video and Object Processor

Overview

The Jaguar video section has been designed to drive a PAL/NTSC TV. The display has a horizontal resolution of up to 720 pixels and a vertical resolution of about 220 lines non-interlaced or 440 lines interlaced. However by adopting a flexible approach to the design the chip can be used with a range of display standards through VGA to Workstation. This will allow the chip to become the backbone of many (possibly unforeseen) products.

Two colour resolutions are supported, 24-bit RGB and our own standard 16-bit CRY (Cyan, Red, Intensity). The 24-bit mode is useful for applications requiring true colour. The 16-bit mode is designed for animation. It consumes less memory, fits better into 64 bit memory, is simpler to shade and is almost indistinguishable from 24-bit mode.

Jaguar decouples the pixel frequency from the system clock by using a line buffer. This means that the system clock does not have to be related to the colour carrier frequency and may be unaffected by gen-locking. There are actually two line buffers one is displayed while the other is prepared by the Object Processor. Each line buffer is a 360 x 32-bit RAM which is cycled at 40 MHz. The line buffer contains physical pixels these may be either 16-bit CRY pixels or 24-bit RGB pixels. The line buffers may be swapped over at the start and in the middle of display lines.

The 16-bit CRY pixels at the output of the line buffer are converted to 24-bit RGB pixels using a combination of look-up tables and small multipliers.

The video timing is completely programmable in units of the pixel clock. The pixel clock can be up to 40 MHz although there is provision for use with an external multiplexer. For TV applications the pixel clock will be in the range 12 to 15 MHz. The pixel clock will be synthesised from the chroma carrier or from an external video source using a device like the MC1378. Eight bits per pixel at up to 160 MHz can be supported by using an external multiplexer, colour-look-up and DAC.

Jaguar uses an Object Processor, this combines the advantages of frame store and sprite based architectures. Jaguar's Object Processor is simple yet sophisticated. It has scaled and unscaled bit-map objects, branch objects for controlling its control flow, and interrupt objects. It can interrupt the graphics processor to perform more complex operations on its behalf. The graphics processor will support perspective, rotation, branches, palette loads, etc.

The Object Processor can write into the line buffer at up to two pixels per clock cycle. The source data can be 1,2,4,8,16 or 24 bits per pixel. Except for 24 bits, objects of different colour resolutions can be mixed. The low resolution objects, one to eight bits, use a palette to obtain a 16-bit physical colour.

A sophistication in the Object Processor is that it can modify the existing contents of the line buffer with another image. This could be used to produce shadows, mist or smoke, coloured glass or say the effect of a room illuminated by flash lamp.

The Object Processor can also ignore data which is stored alongside pixel data if, for instance, a Z buffer is needed then this can be situated next to the pixels. This helps because DRAM RAS pre-charges are needed less frequently.

Object Processor Performance

Each object is described by an object header which is two phrases for an unscaled object and three phrases for a scaled object. When an image has been processed the modified header is written back to memory.

The Object Processor fetches one phrase (64 bits) of video data at a time. This phrase is expanded into pixels (and written into the line buffer) while the next phrase is fetched.

Image data consists of a whole number of phrases. The image data may need to be padded with transparent pixels (colour zero in 1,2,4,8 & 16-bit modes).

The Object Processor writes into the line buffer at one write per system clock tick. In 24-bits-per-pixel mode and for scaled objects one pixel is written per cycle. For unscaled objects with 16 or fewer bits-per-pixel two pixels are written per cycle. Most objects will therefore be expanded at twice the system clock rate.

If the read-modify-write flag is set in the object header the object data is added to the previous contents of the line buffer. In this case the data rate into the line buffer is halved.

This peak rate may be reduced if the memory bandwidth is not high enough. However if 64-bit wide DRAM is installed then these data rates will be sustained for all modes.

When accessing successive locations in 64-bit wide DRAM the memory cycle time is two clock ticks. These are page mode cycles. When the DRAM row address must change there is an overhead of between three and seven clock cycles (depending on DRAM speed). These RAS cycles will occur infrequently during object data fetches but will typically occur during the first data read after reading the object header (because the header and image data will not normally be near each other in memory). RAS cycles will also occur after refresh cycles or if a bus master with a higher priority steals some memory cycles in an area of memory with a different row address. Refresh cycles will normally be postponed until object processing has completed.

Memory controller

Jaguar's memory controller is very fast and flexible. It hides the memory width, speed and type from the other parts of the system.

Memory is grouped into banks that may be of different widths, speeds and types (although both ROM banks have the same width and speed). Each bank is enabled by a chip select. In the case of DRAM there are two chip selects RAS & CAS. Memory widths can be 8,16,32 or 64 bits wide but the memory controller makes it all look 64 bits wide.

There are eight write strobes - one for each eight bits. There are three output enables corresponding to d[0-15],d[16-31] and d[32-63]. Three memory types are supported: DRAM, SRAM and ROM.

ROM or EPROM is used for bootstrap and for cartridges. The ROM speed is programmable. The memory controller allows the system to view ROM as 64 bits wide. Pull-up and pull-down resistors determine the ROM width during reset.

DRAM is the principal memory type, as it is cheap and fast when used in fast page mode. In fast page mode the DRAM cycles at two ticks per transfer. The row time access is programmable. The column access time is not programmable and can only be adjusted by changing the system clock (a page mode cycle takes two clock ticks). The memory controller decides on a cycle by cycle basis whether the next cycle can be a fast page mode cycle. Data and algorithms should be organised to minimise the number of page changes.

There are four memory banks, two of ROM and two of DRAM.

Microprocessor Interface

JAGUAR has been designed to work with any 16 or 32-bit microprocessor with (up to) 24 address lines The interface is based on the 68000 but most microprocessors can be attached by using a PAL to synthesize those control signals which differ All peripherals are memory mapped, there is no separate IO space

The width of the microprocessor is determined during reset by a pull-up / pull-down resistor Variations in the address of the cold boot code/vector is accommodated by making the bootstrap ROM appear everywhere until the memory configuration is set up by the microprocessor

The microprocessor interface is generally asynchronous so the clock speeds of the microprocessor and co-processors may be independent

Jerry uses the same microprocessor interface

The CPU normally has the lowest bus priority but under interrupt its priority is increased

The following list gives the priorities of all bus masters

Highest priority
1. Higher priority daisy-chained bus master
2. Refresh
3. DSP at DMA priority
4. GPU at DMA priority
5. Blitter at high priority
6. Object Processor
7. DSP at normal priority
8. CPU under interrupt
9. GPU at normal priority
10. Blitter at normal priority
11. CPU

Lowest priority

Memory Map

Jaguar's memory map depends on how it is being used

Following reset the following 2 Mbyte window, corresponding to the ROM0 area, is repeated throughout the 16 Mbyte address space until memory is configured by the microprocessor by writing to MEMCON1 (This allows the system to boot whether the microprocessor is a 680X0, an 80X86 or a Transputer.) After configuration, this map corresponds to the area defined as ROM0 on the maps below

Address	Area
1FFFFF	Bootstrap ROM
120000	Jerry DSP
118000	Joysticks and GPIO0-5
114000	Jerry
110000	Internal Registers
100000	Bootstrap ROM
000000	

When the memory configuration is set one of two memory maps is selected depending on bit ROMHI of the memory configuration register

ROMHI=1

Address	Area	Size
FFFFFF	ROM0 Bootstrap ROM and registers	2 Mbytes
E00000	ROM1 Cartridge ROM	6 Mbytes
800000	DRAM1 Dynamic RAM	4 Mbytes
400000	DRAM0 Dynamic RAM	4 Mbytes
000000		

ROMHI=0

Address	Area	Size
FFFFFF	DRAM0 Dynamic RAM	4 Mbytes
C00000	DRAM1 Dynamic RAM	4 Mbytes
800000	ROM1 Cartridge ROM	6 Mbytes
200000	ROM0 Bootstrap ROM and registers	2 Mbytes
000000		

ROM0 is the bootstrap ROM but internal (ASIC) memory and peripherals occupy 128 Kbytes of this space, as shown above. ROM1 is the cartridge ROM DRAM0 and DRAM1 are the two banks of DRAM

A 68000 system will naturally operate with RAM at 0, so the ROMHI map is assumed throughout this document If the system is operated with ROMHI = 0 then the first digit of all internal addresses should be 1 rather than F

Internal Memory Map

Internal Memory is mostly 16 bits wide to allow operation with 16-bit microprocessors

32-bit write cycles are allowed to some areas of internal memory notably the line buffer and the graphics processor memory The line buffer support 32-bit writes primarily in order to accelerate Blitter writes to the line buffer The graphics processor supports 32-bit writes to accelerate program and data loads

MEMCON1 Memory Configuration Register One F00000 RW

Bit 0	ROMHI	When set the two ROM decodes address the top 8M within the 16M window When clear the ROM decodes address the bottom 8M This document assumes throughout that ROMHI is set when discussing register addresses
Bits 1,2	ROMWIDTH	Specifies the width of ROM 0 8 bits 1 16 bits 2 32 bits 3 64 bits
Bits 3,4	ROMSPEED	Specifies the ROM cycle time 0 10 clock cycles 1 8 clock cycles 2 6 clock cycles 3 5 clock cycles
Bits 5,6	DRAMSPEED	Specifies the DRAM Speed The page mode cycle time is always two clock cycles These bits determine RAS related timing as follows <table><tr><td>Bits 5,6</td><td>Precharge</td><td>RAS to CAS</td><td>Refresh</td></tr><tr><td>0</td><td>4</td><td>3</td><td>5</td></tr><tr><td>1</td><td>4</td><td>3</td><td>4</td></tr><tr><td>2</td><td>3</td><td>2</td><td>4</td></tr><tr><td>3</td><td>2</td><td>1</td><td>3</td></tr></table>The times are clock cycles
Bit 7	FASTROM	Sets the ROM cycle time to two clock cycles This is for test purposes only
Bits 8-10	unused	Set to zero
Bits 11,12	IOSPEED	Specifies the speed of external peripherals The number of cycles here is the overall cycle time, the control strobes are active for two cycles less than this 0 18 clock cycles 1 10 clock cycles 2 4 clock cycles 3 6 clock cycles
Bit 13	unused	Set to zero
Bit 14	CPU32	Indicates that the microprocessor is 32 bits
Bit 15	unused	Set to zero

All the ROMSPEED bits are set to zero on reset. ROMHI, ROMWIDTH and CPU32 are determined by external pull-up / pull-down resistors. All the other bits are undefined. ROM0 repeats every 2 Mbytes until this register is written to.

MEMCON2 Memory Configuration Register Two F00002 RW

Bits 0,1	COLS0	Specifies number of columns in DRAM0 0 256 1 512 2 1024 3 2048
Bits 2,3	DWIDTH0	Specifies the width of DRAM0 0 8 bits 1 16 bits 2 32 bits 3 64 bits
Bits 4,5	COLS1	Specifies number of columns in DRAM1 0 256 1 512 2 1024 3 2048
Bits 6,7	DWIDTH1	Specifies the width of DRAM1 0 8 bits 1 16 bits 2 32 bits 3 64 bits
Bits 8-11	REFRATE	Specifies the refresh rate. DRAM rows are refreshed at a frequency of CLK / (64 x (REFRATE+1)). Many DRAM chips require a refresh frequency of 64 KHz. Refresh cycles occur at the end of object processing. If REFRATE is zero refresh is disabled.
Bit 12	BIGEND	Specifies that big-endian addressing should be used. This determines the address of a byte within a phrase and allows Jaguar to be used comfortably with Big-endian (Motorola) processors or with Little-endian (Intel) processors.
Bit 13	HILO	Specifies that image data should be displayed from high order bits to low order.

All the above bits are undefined on reset except BIGEND which is determined by external pull-up / pull-down resistors.

HC Horizontal Count F00004 RW

This register comprises of a ten bit counter which counts from zero up to the value in the horizontal period register twice per video line. An eleventh bit determines which half of the display is being generated. The counter is incremented by the pixel clock. The vertical counter is incremented every half line in order to support interlaced displays. This register is only for ASIC test purposes.

VC	Vertical Count	F00006	RW

This register comprises of an eleven bit counter which counts from zero up to the value in the vertical period register once per field. A twelfth bit determines which field (odd/even) is being generated. The counter is incremented every half line. This register can be read to do beam synchronous operations. It is only written to for ASIC test purposes.

LPH	Horizontal Light-pen	F00008	RO

This read only eleven bit register gives the horizontal position in pixels of the light-pen.

LPV	Vertical Light-pen	F0000A	RO

The low eleven bits of this register gives the vertical position of the light-pen in half lines.

OB[0-3]	Object Code	F00010-16	RO

These four registers allow the graphics processor to read the current object. This allows the graphics processor object to pass parameters to the GPU interrupt service routine.

OLP	Object List Pointer	F00020	WO

This 32-bit register points to the start of the object list. All objects must be on a phrase boundary so the bottom three bits are always zero. When one object links to another bits 3 to 21 of this address are replaced by the LINK data in the object.

OBF	Object Processor flag	F00026	WO

Bit zero of this register can be tested by the Object Processor branch instruction. If set the branch is taken, if clear execution continues with the next object. This flag is intended as a mechanism for letting the graphics processor control the Object Processor program flow. A write (of anything) to this register restarts the Object Processor after a Graphics Processor interrupt object.

VMODE	Video Mode	F00028	WO

Bit 0	VIDEN	When set enables time-base generator
Bits 1,2	MODE	Determines how the line buffer contents are translated into physical pixels.
	0	16-bit CRY. Each 32-bit entry in the line buffer is treated as two 16-bit CRY pixels on successive clock cycles. Each is converted into eight bits of red, green & blue using a combination of lookup tables and multipliers.
	1	24-bit RGB. Each 32-bit entry in the line buffer is treated as one physical pixel with eight bits of red, eight bits of blue, eight bits of green and eight bits unused.

	2	16-bit direct. Each 32-bit entry in the line buffer is divided into two 16-bit words which are output directly onto the red and green outputs on alternate phases of the video clock. This mode is for applications requiring a dot clock in excess of 40 MHz. It is assumed that further multiplexing and colour lookup will occur outside the chip. In this mode blanking and video active are output on the two least significant bits of blue.
	3	16-bit RGB. Each 32-bit entry in the line buffer is treated as two 16-bit RGB pixels. Bits [0-5] are green, bits [6-10] are blue and bits [11-15] are red.
Bit 3	GENLOCK	When set this bit enables digital genlocking. This means that external syncs will reset the internal time-base generators. On its own this mechanism does not give satisfactory genlocking because there is a jitter of up to one pixel. However this mechanism is used to quickly lock onto a new video source. An external Phase Locked Loop is required for true genlocking.
Bit 4	INCEN	Enables encrustation. When set the least significant bit of the CRY intensity is used to switch between local and external video sources using an external video multiplexer. This allows the video source to be switched on a pixel by pixel basis.
Bit 5	BINC	Selects the local border colour if encrustation is enabled.
Bit 6	CSYNC	Enables composite sync on the vertical sync output.
Bit 7	BGEN	Clears the line buffer to the colour in the background register after displaying the contents. This only has effect in CRY and RGB16 modes.
Bit 8	VARMOD	Enables variable colour resolution mode. When this bit is set the least significant bit of each word in the line buffer is used to determine the colour coding scheme of the other 15 bits. If the bit is clear the bits the word is treated as a CRY pixel. If the bit is set then bits [1-5] are green, bits [6-10] are blue and bits [11-15] are red. This mechanism allows JAGUAR to support an RGB window against a CRY background for instance.
Bits 9-11	PWIDTH	This field determines the width of pixels in video clock cycles. The width is one more than the value in this field. The video time base generator is programmed in cycles of the video clock and not the pixel clock produced by this divider. The display width should be set to be an integer number of pixels, i.e. an integer multiple of the pixel width programmed here.
Bits 12-15	Unused	Write zeroes.

BORD1	**Border Colour (Red & Green)**	**F0002A**	**WO**
BORD2	**Border Colour (Blue)**	**F0002C**	**WO**

These registers determine the physical border colour. There are eight bits per primary colour. Red is the less significant byte of BORD1. This colour is displayed between the active portions of the screen and blanking. It is not necessary to display a border. The border area is defined by the video time-base registers.

HP	Horizontal Period	F0002E	WO

This ten bit register determines the period of half a display line in video clock cycles The period is one tick longer than the value written into this register

HBB	Horizontal Blanking Begin	F00030	WO

This eleven bit register determines the start position of horizontal blanking The most significant bit is usually set because blanking starts in the second half of the line

HBE	Horizontal Blanking End	F00032	WO

This eleven bit register determines the end position of horizontal blanking The most significant bit is usually clear because blanking ends in the first half of the line

HS	Horizontal Sync	F00034	WO

This eleven bit register determines the width of the horizontal sync and equalization pulses The pulses start when the horizontal count equals the value in the register The pulses end when the horizontal count equals the horizontal period. The most significant bit is usually set because horizontal sync happens at the end of the line The most significant bit is ignored in the generation of equalization pulses which are the same width as horizontal sync but which appear twice per line (for 10 half lines during field blanking)

HVS	Horizontal Vertical Sync	F00036	WO

This ten bit register determines the end position of the vertical sync pulses Vertical Sync consists of long sync pulses for several half lines These pulses are generated twice per line Vertical sync starts at the same time as the horizontal sync or equalization pulses but end when the least significant ten bits of the horizontal count match the HVS register

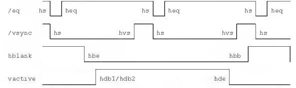
HDB1	Horizontal Display Begin 1	F00038	WO
HDB2	Horizontal Display Begin 2	F0003A	WO

These eleven bit registers control where on the display line the Object Processor starts When the horizontal count matches either of the above registers the Object Processor starts execution at the address in OLP, the line buffers swap over and pixels are shifted out of the line buffer The Object Processor can run twice per line in order to support display modes where the amount of data on a display line is greater than can be contained in one line buffer The line buffers are each 360 words x 32 bits If the display mode was 720 x 24 bits per pixel then line buffer A might be displayed at the start of the line while buffer B was being written Then during the second half of the display line buffer B would be displayed while line buffer A was prepared for the next line In this case HDB1 would contain a value corresponding to the left hand edge of the display and HDB2 would contain a value corresponding to the middle of the display If the Object Processor needs to run only once per line then either the registers take the same value or one register is given a value greater than the line length

HDE	Horizontal Display End	F0003C	WO

This eleven bit register specifies when the display ends Either border colour or black (if HBB < HDE) is displayed after the horizontal count matches this register

The relative positions of some of the above signals and the registers which define them are shown on the following diagram.



VP Vertical Period F0003E WO

This eleven bit register determines the number of half lines per field. The number is one more than the value written into this register. If the number of half lines is odd then the display is interlaced.

VBB Vertical Blanking Begin F00040 WO

This eleven bit register specifies the half line on which vertical blanking begins.

VBE Vertical Blanking End F00042 WO

This eleven bit register specifies the half line on which vertical blanking ends.

VS Vertical Sync F00044 WO

This eleven bit register specifies the half line on which vertical sync begins. Vertical sync pulses are generated from this line to the line specified by the vertical period.

VDB Vertical Display Begin F00046 WO

This eleven bit register specifies the half line on which object processing begins. Object processing restarts on every line until the half line specified by the VDE register. The border colour (or black) is displayed outside these active lines.

VDE Vertical Display End F00048 WO

This eleven bit register specifies the half line at which object processing ends.

VEB Vertical Equalization Begin F0004A WO

This eleven bit register specifies the half line on which equalization pulses start

VEE Vertical Equalization End F0004C WO

This eleven bit register specifies the half line on which equalization pulses end

VI Vertical Interrupt F0004E WO

This eleven bit register specifies a half line on which the VI interrupt is generated. This number must be odd for non-interlaced setups

PIT[0-1] Programmable Interrupt Timer F00050-52 WO

These two 16-bit registers control the frequency of interrupts to the CPU and to the GPU. PIT[0] & PIT[1] operate as a pair controlling the interrupts

The system clock is divided by (one plus the value in the first register). If the first register contains zero the timer is disabled. The resulting frequency is divided by (one plus the value in the second register) and the output of this divider generates the interrupt

HEQ Horizontal equalization end F00054 WO

This ten bit register determines the end position of the equalization pulses. Equalization consists of short sync pulses for several half lines on either side of vertical sync. These pulses are generated twice per line.

BG Background Colour F00058 WO

This register specifies the CRY colour to which the line buffer is cleared

INT1 CPU Interrupt Control Register F000E0 RW

This register enables, identifies and acknowledges interrupts from the five different CPU interrupt sources. The interrupts sources are as follows

0	Video	This interrupt is generated by the video time-base, on a line selected by the VI register
1	GPU	This interrupt is generated by the graphics processor writing to an internal register
2	Object	This interrupt is generated by stop objects
3	Timer	This interrupt is generated by the programmable timer (PIT) in TOM
4	Jerry	This interrupt is generated by an input to Tom and is intended for use by Jerry. This is an active high edge-triggered interrupt - the first interrupt will occur on the first rising edge after it has been enabled.

Bits 0 to 4 enable the individual interrupt sources, i.e. if bit 1 is set the graphics processor interrupt is enabled. When read bits 0 to 4 indicate which interrupts are pending, i.e. if bit 3 is set there is an timer interrupt pending. Bits 8 to 12 clear pending interrupts from the corresponding interrupt source.

Note that INT2 must always be written to at the end of a CPU interrupt service routine

INT2	CPU Interrupt resume register	F000E2	WO

When an interrupt is applied to the CPU the bus priorities of the graphics processor and Blitter are reduced so that the CPU can service real time interrupts promptly The bus priorities are restored by writing any value to this register This should therefore always be done at the end of an interrupt service routine After the write to this port the Blitter or GPU may then restart, and no further instructions will then be executed until either the next interrupt occurs, or the GPU or Blitter operation completes

CLUT	Colour Look-Up Table	F00400-7FE RW

The colour look-up table translates an eight bit colour index into a 16-bit physical colour (CRY or 16-bit RGB) The eight bit index comes from the object data, which may be 1,2,4 or 8 bits In order to achieve a high throughput there are two tables allowing two pixels at a time to be written into the line buffer There are 256 16-bit entries in each table Locations in the range F00400-5FE read from table A Addresses in the range F00600-7FE read from table B Writing to either address range writes to both tables

LBUF	Line Buffer	F00800-0D9E F01000-159E F01800-1D9E	RW

There are two line buffers each of which consists of a 360 x 32-bit RAM Each 32-bit long-word can be read/written as two 16-bit words In 16-bit CRY mode each word is a CRY pixel; the less significant byte is the intensity The word with the lowest address corresponds to the left-most pixel In 24-bit RGB mode each 32-bit long-word is a pixel The less significant byte of the word at the lower address is the red value The more significant byte is the green value and the less significant byte of the word at the high address is the blue value The fourth byte is unused

The first address range addresses line buffer A The second addresses line buffer B The third addresses the line buffer currently selected for writing The first two address ranges are for test purposes the third is for the graphics processor to assist the Object Processor in preparing the line buffer

By adding 8000h to the above address ranges 32-bit writes can be made to the line buffer This is mainly to accelerate the Blitter

Peripheral Memory Map

Jerry and external peripherals occupy the 64k above the internal memory All Peripheral Memory is 16 bits wide although it is likely that many devices will have eight bit busses

Object definitions

There are five basic object types

Bit Mapped Object

This object displays an unscaled bit mapped object The object must be on a 16 byte boundary in 64 bit RAM

First Phrase

Bits	Field	Description
0-2	TYPE	Bit mapped object is type zero
3-13	YPOS	This field gives the value in the vertical counter (in half lines) for the first (top) line of the object The vertical counter is latched when the Object Processor starts so it has the same value across the whole line If the display is interlaced the number is even for even lines and odd for odd lines If the display is non-interlaced the number is always even The object will be active while the vertical counter >= YPOS and HEIGHT > 0
14-23	HEIGHT	This field gives the number of data lines in the object As each line is displayed the height is reduced by one for non-interlaced displays or by two for interlaced displays (The height becomes zero if this would result in a negative value) The new value is written back to the object
24-42	LINK	This defines the address of the next object These nineteen bits replace bits 3 to 21 in the register OLP This allows an object to link to another object within the same 4 Mbytes
43-63	DATA	This defines where the pixel data can be found Like LINK this is a phrase address These twenty-one bits define bits 3 to 23 of the data address This allows object data to be positioned anywhere in memory After a line is displayed the new data address is written back to the object

Second Phrase

Bits	Field	Description
0-11	XPOS	This defines the X position of the first pixel to be plotted This 12 bit field defines start positions in the range -2048 to +2047 Address 0 refers to the left-most pixel in the line buffer
12-14	DEPTH	This defines the number of bits per pixel as follows
		0 1 bit/pixel
		1 2 bits/pixel
		2 4 bits/pixel
		3 8 bits/pixel
		4 16 bits/pixel
		5 24 bits/pixel
15-17	PITCH	This value defines how much data, embedded in the image data, must be skipped For instance two screens and their common Z buffer could be arranged in memory in successive phrases (in order that access to the Z buffer does not cause a page fault) The value 8 * PITCH is added to the data address when a new phrase must be fetched A pitch value of one is used when the pixel data is contiguous - a value of zero will cause the same phrase to be repeated

18-27	DWIDTH	This is the data width in phrases i.e. Data for the next line of pixels can be found at 8 * (DATA + DWIDTH)
28-37	IWIDTH	This is the image width in phrases (must be non zero), and may be used for clipping
38-44	INDEX	For images with 1 to 4 bits/pixel the top 7 to 4 bits of the index provide the most significant bits of the palette address
45	REFLECT	Flag to draw object from right to left
46	RMW	Flag to add object to data in line buffer. The values are then signed offsets for intensity and the two colour vectors
47	TRANS	Flag to make logical colour zero and reserved physical colours transparent
48	RELEASE	This bit forces the Object Processor to release the bus between data fetches. This should typically be set for low colour resolution objects because there is time for another bus master to use the bus between data fetches. For high colour resolution objects the bus should be held by the Object Processor because there is very little time between data fetches and other bus masters would probably cause DRAM page faults thereby slowing the system. External bus masters, the refresh mechanism and graphics processor DMA mechanism all have higher bus priorities and are unaffected by this bit
49-54	FIRSTPIX	This field identifies the first pixel to be displayed. This can be used to clip an image. The significance of the bits depends on the colour resolution of the object and whether the object is scaled. The least significant bit is only significant for scaled objects where the pixels are written into the line buffer one at a time. The remaining bits define the first pair of pixels to be displayed. In 1 bit per pixel mode all five bits are significant, In 2 bits per pixel mode only the top four bits are significant. Writing zeroes to this field displays the whole phrase.
55-63		Unused write zeroes.

Scaled Bit Mapped Object

This object displays a scaled bit mapped object. The object must be on a 32 byte boundary in 64 bit RAM. The first 128 bits are identical to the bit mapped object except that TYPE is one. An extra phrase is appended to the object.

Bits	Field	Description
0-7	HSCALE	This eight bit field contains a three bit integer part and a five bit fractional part. The number determines how many pixels are written into the line buffer for each source pixel
8-15	VSCALE	This eight bit field contains a three bit integer part and a five bit fractional part. The number determines how many display lines are drawn for each source line. This value equals HSCALE for an object to maintain its aspect ratio
16-23	REMAINDER	This eight bit field contains a three bit integer part and a five bit fractional part. The number determines how many display lines are left to be drawn from the current source line. After each display line is drawn this value is decremented by one. If it becomes negative then VSCALE is added to the remainder until it becomes positive. HEIGHT is decremented every time VSCALE is added to the remainder. The new REMAINDER is written back to the object

24-63	Unused write zeroes.

Graphics Processor Object

This object interrupts the graphics processor, which may act on behalf of the Object Processor. The Object Processor resumes when the graphics processor writes to the object flag register.

Bits	Field	Description
0-2	TYPE	GPU object is type two
3-13	YPOS	This object is active when the vertical count matches YPOS unless YPOS = 07FF in which case it is active for all values of vertical count
14-63	DATA	These bits may be used by the GPU interrupt service routine. They are memory mapped as the object code registers OB0-3, so the GPU can use them as data or as a pointer to additional parameters.

Execution continues with the object in the next phrase. The GPU may set or clear the (memory mapped) Object Processor flag and this can be used to redirect the Object Processor using the following object.

Branch Object

This object directs object processing either to the LINK address or to the object in the following phrase.

Bits	Field	Description
0-2	TYPE	Branch object is type three
3-13	YPOS	This value may be used to determine whether the LINK address is used
14-15	CC	These bits specify what condition is used to determine whether to branch as follows: 0 Branch if YPOS == VC or YPOS == 7FF 1 Branch if YPOS > VC 2 Branch if YPOS < VC 3 Branch if Object Processor flag is set 4 Branch if on second half of display line (HC10 = 1)
16-23	unused	
24-42	LINK	This defines the address of the next object if the branch is taken. The address is defined as described for the bit mapped object.
43-63	unused	

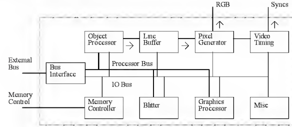
Stop Object

This object stops object processing and interrupts the host.

Bits	Field	Description
0-2	TYPE	Stop object is type four
3-63	DATA	These bits may be used by the CPU interrupt service routine. They are memory mapped so the CPU can use them as data or as a pointer to additional parameters.

Description of Object Processor/Pixel path

The following two diagrams show where the object data path fits into the TOM chip. All the diagrams that follow are drastically simplified for clarity.

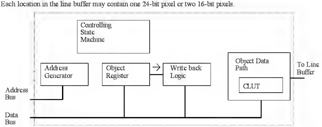


Jaguar Chip Block Diagram

The processor bus is a 64-bit data, 24-bit address multi-master bus. The bus master can change on a cycle by cycle basis with no overhead. The external CPU controls this bus when it is the bus master. The IO bus is a 16 data 16 address bus used for reading and writing to internal memory and registers. The bus interface logic and memory controller allows transfers of any width (one to eight bytes) to be made to any width of external memory. The bus interface accommodates 16 and 32-bit microprocessors. The bus interface also generates a multiplexed address for dynamic RAMs. The multiplexed address is a function of memory width and number of columns. The memory controller only performs RAS cycles when the row address changes. Thus allows contiguous regions of memory to be accessed much faster.

The line buffer is a bridge between two asynchronous parts of the chip. On one side are the processors and memory. On the other side are the video timing and pixel generators. In fact there are two line buffers. While one is written into by the Object Processor, the other is read by the pixel logic. Each line buffer is a small 360x32 RAM with independent write strobes for the high and low words.

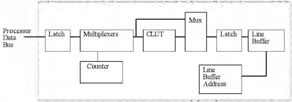
Each location in the line buffer may contain one 24-bit pixel or two 16-bit pixels.



Object Processor Block Diagram

The Object Processor reads object headers and image data and writes back modified headers. The write back logic normally increases the data address by the data width. If the object is scaled then the data address is increased by a multiple of the data width and the vertical remainder is modified.

The object data contains either physical colours in the case of 16 and 24 bits-per-pixel objects or logical colours in the case of 1,2,4 and 8 bits-per-pixel objects. Logical colours are translated into physical colours by the colour look up table or CLUT.



Object Data Path

The Object Processor fetches data one phrase at a time until the image data, for that header, is exhausted or until the line buffer address (X co-ordinate) has become invalid. The behaviour of the object data path depends on the colour resolution of the object (bits-per-pixel) and on whether the object is scaled.

In 24 bits-per-pixel mode each phrase contains two pixels (16 bits unused per phrase). The multiplexers select each in turn and one 24-bit pixel is written into the line buffer per clock cycle. The CLUT is bypassed for 24 bits-per-pixel objects.

In 16 bits-per-pixel mode each phrase contains four pixels. The multiplexers select two pixels at a time and two pixels are written into the line buffer each clock cycle. The CLUT is bypassed for 16 bits-per-pixel objects.

In 1, 2, 4 and 8 bits-per-pixel modes each phrase contains 64, 32, 16 and 8 pixels respectively. The multiplexers select two pixels at a time. In 1, 2 and 4 bit modes the pixel is made up to eight bits by taking the top bits from the top bits of the palette offset (a field in the object header). The two eight bit values are used as addresses to a pair of identical CLUTs yielding two sixteen bit physical pixels which are written into the line buffer every cycle.

If an object is scaled the Object Processor deals with one pixel at a time not pairs. Scaling is achieved by incrementing the line buffer address independently of the counter controlling the multiplexer. For instance if the line buffer address is incremented twice as often as the counter then the image will be twice as wide.

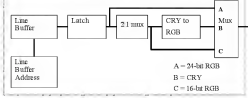
There are two line buffers A & B. While A is written by the Object Processor B is being read by the pixel logic. At the start of the next display line the buffers swap over so A is displayed and B is written. This swap is effectively achieved by multiplexers on all the signals attached to the line buffers.

The above description is complicated by the following:

- If a pair of pixels must be written to an odd location in the line buffer they must be swapped and one pixel delayed.
- The line buffer address decrements if the object is reflected.
- The colour to be written into the line buffer can be added to the previous value instead.
- One colour may be used as transparent and is not written into the line buffer.

- The line buffers also appear as memory to the rest of the system

The pixel data path is shown in the following diagram All the logic in this box runs from a different clock to the previous logic, this is the video clock



Pixel Data Path

The operation of the pixel data path depends on the video mode.

In 24 bits-per-pixel mode the line buffer is read at the video clock frequency The line buffer data is simply latched and presented at the pins as red, green and blue data bits

In CRY mode the line buffer is read at half the video clock frequency Each read yields two 16-bit CRY values These are multiplexed into the CRY to RGB conversion logic during succeeding video clock cycles In this logic the more significant eight bits specify the colour and the less significant bits specify the intensity or brightness The colour value is used as an index to three ROMs These ROMs contain the relative amounts of red, green and blue for each colour The outputs of the ROMs are multiplied by the brightness to get a final eight bits of red, green and blue

In RGB16 mode the line buffer is read at half the video clock frequency Each read yields two 16-bit RGB values Bits 0-5 form the six most significant bits of green, bits 6-10 form the five most significant bits of blue and bits 11-15 form the five most significant bits of red All other bits are set to zero.

In all these modes a small amount of additional logic sets the output colour to black during blanking and to the border colour where appropriate

A fourth mode exists to allow the system to support very high pixel rates using external multiplexers and DACs This is called direct mode In this mode the line buffer is read at the video clock frequency and the 2 1 multiplexer is driven by the video clock directly The output of the 2 1 mux is connected directly to the red and green outputs of the chip This allows 16-bit values to be output at twice the maximum video clock frequency This provides a video bandwidth of up to 4 times the video clock rate (in bytes per second) These values should be re-synchronised, de-multiplexed and converted to analogue outside the chip In this mode the blanking and border signals are output on the blue pins

The above picture is slightly complicated by the following

- The least significant bit in CRY and RGB16 modes can be sacrificed (treated as zero) and used to control an external video switch through the incrust output pin
- In CRY and RGB16 modes a background colour may be written into the line buffer after it has been read
- In CRY and RGB16 modes the least significant bit may be used to determine whether the mode is CRY or RGB16 This could be used to drop a decompressed RGB picture into a CRY picture without having to do a RGB to CRY conversion

Refresh Mechanism

The average refresh frequency is defined by the REFRATE bits in the MEMCON2 register Refresh cycles are grouped together in order to lessen the impact on system performance However they cannot be performed in very large numbers or they would create "dead spots" in which no processing was possible This could disrupt the display or sound production

Jaguar uses a counter to accumulate a count of refresh cycles When this counter reaches eight then eight refresh cycles are done and the counter is set to zero

Refresh cycles are also invoked when the Object Processor reaches the end of the object list After the Object Processor executes a STOP object JAGUAR performs as many refresh cycles as are necessary to decrement the refresh counter to zero

This mechanism guarantees that the minimum refresh rate is maintained without interrupting the Object Processor and without creating "dead spots" of more than a few microseconds

Colour Mapping

Introduction

Jaguar produces a video output using eight digital bits each for red, green and blue. This allows each output to have two hundred and fifty-six intensity levels, and is enough to allow smooth shading from one colour to another. This twenty-four bit scheme is known as *true-colour*.

Jaguar can produce a display based on true colour pixels stored in memory in long words, with eight bits unused, and this is known as true colour mode. However, these thirty-two bit pixels are large and so consume a lot of memory; and they also consume a lot of memory bandwidth to fetch from RAM for display.

True-colour mode is therefore unattractive for general use, as most images do not need its range of colours, and it is desirable to avoid the detrimental effects it has on performance. True colour mode is therefore a special case, and when it is used only true-colour images may be displayed.

In normal operation, the Jaguar display system is based on sixteen-bit pixels. Images in memory may be stored either as sixteen bit pixels, or as one, two, four or eight bit *logical* colours. These logical colours are used as indices into a Palette or Colour-Look-Up-Table (CLUT), which contains their corresponding sixteen-bit physical colours.

Sixteen-bit pixels may be stored as six bits of green, and five bits each for red and blue, but this no longer allows smooth shading. There is therefore an additional scheme, known as the CRY scheme (cyan, red and intensity, see below) which still allows smooth intensity shading. This CRY scheme is now discussed in greater detail.

The CRY Colour Scheme

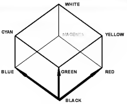
Gouraud Shading Requirements

The CRY scheme was derived principally to meet the requirements of *Gouraud Shading*. This is a technique that models the appearance of a lit curved surface from a set of polygons. The problem the technique helps to overcome is that if the intensity due to a light source is calculated for each polygon and the polygon is painted in that colour, then the polygons that make up that surface are each clearly visible.

The technique of Gouraud shading helps avoid this by calculating the intensity at each vertex, and then linearly interpolating along each polygon edge, and hence along each scan line that makes up the display. If only white light sources are considered, then the only variation is one of luminous intensity, and not one of colour. It is therefore attractive to have a colour scheme that contains an intensity vector, as the Gouraud shading calculations have then only to be performed for one value, rather than the three values that would have to be calculated in a true colour scheme.

As there is general agreement that eight bits is enough to give smooth intensity shading (and it is a round number), it was therefore necessary to come up with a scheme that allowed the colour to be expressed in eight bits.

Colour Space



The colour space to be modelled may be considered as the RGB cube shown, where the lowest vertex represents black, and the highest white. The three edges running out from black are the three orthogonal vectors red, green and blue. The sum of these three vectors can describe any point in the cube. The three lower vertices therefore represent fully saturated red, green and blue, and the three higher ones yellow, cyan and magenta.

This colour space model is only one of many ways of considering what the human brain 'sees', but it has the advantage of modelling the display system used by colour monitors, and of being mathematically simple.

Physical requirements

The intensity vector can be considered as that component of the sum of the red, green and blue vectors that lies along the diagonal of the RGB cube from black to white. This is not the 'true' intensity, which is a weighted sum of red, green, and blue, but it bears a linear relationship to it when the colour is not changed.

It is necessary to come up with a scheme to encode the colour value in the remaining eight bits of the pixel. The following requirements were made on this scheme:

1. All two hundred and fifty-six values should represent valid, and different, colours.
2. The colours should be well spread out across the colour space.
3. Colours should be able to be mixed by linearly averaging their colour values.
4. An intensity value of zero must be black.

As the remaining colour space without intensity is two-dimensional, two vectors are required to represent a point in it. An ***r, theta*** scheme was discarded as it would not meet requirement two, and so a scheme based on two ***x, y*** vectors was chosen.

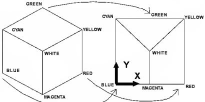
To meet requirement one, the two vectors must describe a point on a square area. As no existing colour space model is square when viewed along the intensity axis, it was necessary to come up with a new one.

The approach chosen, after considerable experimentation, was to take the view along the intensity axis of the RGB cube, which is a hexagon, and distort it into a square. This does not quite meet requirement 3, but is close to it.

CRY Colour Scheme

The colour mapping scheme chosen is based on defining 256 points on the upper surface of the RGB cube.

In the figure shown, the hexagon corresponds to a view looking down onto the RGB cube. This hexagon is distorted onto a square, whose X and Y co-ordinates are four-bit values. This defines 256 colour levels. The choice of green as the primary colour that lies on the middle of one face was made after observing the effects of the three possible mappings, and corresponds with the expected result, as the human eye is least able to distinguish shades of green.



Note that in each of the three areas defined on the hexagon and square, one of red, green or blue is at full intensity, and the others vary. At the centre (white) they are all at full intensity. The intensity scale for any given colour lies along the line between black, and the point on the top surface of the cube defined in the colour table.

Colours may be averaged by taking the average of their eight-bit intensity value, and each of the four-bit X and Y components of the colour value. This will not produce exactly the same colour as the point midway between them in the RGB cube, but will be close to it.

This is a summary of the pros and cons of the CRY scheme:

Advantages of CRY

- Smooth intensity shading from 16-bit pixels
- Better matched to the capabilities of the human eye than 5:6:5 bit RGB schemes
- Suitable for efficient Gouraud shading

Disadvantages

- Steps are visible in smooth changes of saturation or hue
- Translation from RGB to CRY is not straightforward
- Non-standard

RGB to CRY Conversion

The best technique is to calculate the intensity value, which is the largest of red, green and blue, and from this the ideal ROM entry for that colour, by scaling the RGB values by 255 / intensity. This can then be matched to the actual ROM tables to find the nearest match. A quick way of doing this is by a lookup table. It is not necessary for this to have 2^{24} entries, it turns out that taking the top 5 bits of each of the red, green and blue values (rounding where appropriate) and using a 32768 element lookup table is adequate.

Physical Implementation

The eight-bit colour value is used to index a look-up table of modifier values for each of red green and blue, which is multiplied by the intensity value to give the output level for each drive to the display. The look-up tables are:

RED	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	34	34	34	34	34	34	34	34	34	34	34	34	34	34	19	0
	68	68	68	68	68	68	68	68	68	68	68	68	64	43	21	0
	102	102	102	102	102	102	102	102	102	102	102	95	71	47	23	0
	135	135	135	135	135	135	135	135	135	135	130	104	78	52	26	0
	169	169	169	169	169	169	169	169	169	170	141	113	85	56	28	0
	203	203	203	203	203	203	203	203	203	183	153	122	91	61	30	0
	237	237	237	237	237	237	237	237	230	197	164	131	98	65	32	0
	255	255	255	255	255	255	255	255	247	214	181	148	115	82	49	17
	255	255	255	255	255	255	255	255	255	235	204	173	143	112	81	51
	255	255	255	255	255	255	255	255	255	255	227	198	170	141	113	85
	255	255	255	255	255	255	255	255	255	255	249	223	197	171	145	119
	255	255	255	255	255	255	255	255	255	255	255	248	224	200	177	153
	255	255	255	255	255	255	255	255	255	255	255	255	252	230	208	187
	255	255	255	255	255	255	255	255	255	255	255	255	255	255	240	221
	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
GREEN	0	17	34	51	68	85	102	119	136	153	170	187	204	221	238	255
	0	19	38	57	77	96	115	134	154	173	192	211	231	250	255	255
	0	21	43	64	86	107	129	150	172	193	215	236	255	255	255	255
	0	23	47	71	95	119	142	166	190	214	238	255	255	255	255	255
	0	26	52	78	104	130	156	182	208	234	255	255	255	255	255	255
	0	28	56	85	113	141	170	198	226	255	255	255	255	255	255	255
	0	30	61	91	122	153	183	214	244	255	255	255	255	255	255	255
	0	32	65	98	131	164	197	230	255	255	255	255	255	255	255	255
	0	32	65	98	131	164	197	230	255	255	255	255	255	255	255	255
	0	30	61	91	122	153	183	214	244	255	255	255	255	255	255	255
	0	28	56	85	113	141	170	198	226	255	255	255	255	255	255	255
	0	26	52	78	104	130	156	182	208	234	255	255	255	255	255	255
	0	23	47	71	95	119	142	166	190	214	238	255	255	255	255	255
	0	21	43	64	86	107	129	150	172	193	215	236	255	255	255	255
	0	19	38	57	77	96	115	134	154	173	192	211	231	250	255	255
	0	17	34	51	68	85	102	119	136	153	170	187	204	221	238	255
BLUE	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
	255	255	255	255	255	255	255	255	255	255	255	255	255	255	240	221
	255	255	255	255	255	255	255	255	255	255	255	255	252	230	208	187
	255	255	255	255	255	255	255	255	255	255	255	248	224	200	177	153
	255	255	255	255	255	255	255	255	255	255	249	223	197	171	145	119
	255	255	255	255	255	255	255	255	255	255	227	198	170	141	113	85
	255	255	255	255	255	255	255	255	255	235	204	173	143	112	81	51
	255	255	255	255	255	255	255	255	247	214	181	148	115	82	49	17
	237	237	237	237	237	237	237	237	230	197	164	131	98	65	32	0
	203	203	203	203	203	203	203	203	203	183	153	122	91	61	30	0
	169	169	169	169	169	169	169	169	169	170	141	113	85	56	28	0
	135	135	135	135	135	135	135	135	135	135	130	104	78	52	26	0
	102	102	102	102	102	102	102	102	102	102	102	95	71	47	23	0
	68	68	68	68	68	68	68	68	68	68	68	68	64	43	21	0
	34	34	34	34	34	34	34	34	34	34	34	34	34	34	19	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Graphics Processor Subsystem

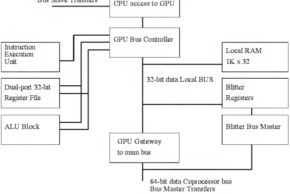
The Graphics Subsystem of Jaguar is a self-contained processing unit, whose view of the external system processor and memory are controlled by a separate memory controller, which is not part the graphics system.

The graphics subsystem transfers data to or from external memory by becoming the master of the co-processor bus. This bus has a 64-bit (phrase) data path, and a 24-bit address, with byte resolution. This bus has multiple masters, and ownership of it is gained by a bus request/acknowledge system, which is prioritised, i.e. ownership can be lost during a request (but not during a memory cycle). The graphics subsystem actually contains two bus masters, the Graphics Processor and the Blitter.

The graphics subsystem also acts as a slave on the IO bus. This bus normally has a 16-bit data path, and allows external processors to access memory and registers within the graphics subsystem. As the data path within the graphics subsystem is 32-bit, all reads and writes must be in pairs.

The memory within the Graphics Subsystem appears to be part of the general machine address space, both to the GPU and Blitter, and to external processors. The advantage to the GPU of having local memory is both that it is faster, and that it does not require ownership of the system bus to be accessed.

This diagram shows the architecture and data paths of the graphics subsystem.



Memory Map

The Graphics sub-system address space contains the following locations

F02100	GPU_FLAGS	RW	GPU flags
F02104	GPU_MTXC	W	GPU matrix control
F02108	GPU_MTXA	W	GPU matrix address
F0210C	GPU_BIGEND	W	GPU big / little endian control
F02110	GPU_PC	RW	GPU program counter
F02114	GPU_CTRL	RW	GPU operation control / status
F02118	GPU_HIDATA	RW	GPU bus interface high data
F0211C	GPU_REMAIN	R	GPU division remainder
F02200	BLIT_A1BASE	W	Blitter A1 base
F02204	BLIT_A1FLAGS	W	Blitter A1 flags
F02208	BLIT_A1WIN	W	Blitter A1 window size
F0220C	BLIT_A1PTR	RW	Blitter A1 pointer
F02210	BLIT_A1STEP	W	Blitter A1 step
F02214	BLIT_A1STEPF	W	Blitter A1 step fraction
F02218	BLIT_A1FRAC	RW	Blitter A1 pointer fraction
F0221C	BLIT_A1INC	W	Blitter A1 pointer increment
F02220	BLIT_A1INCF	W	Blitter A1 pointer increment fraction
F02224	BLIT_A2BASE	W	Blitter A2 base
F02228	BLIT_A2FLAGS	W	Blitter A2 flags
F0222C	BLIT_A2MASK	W	Blitter A2 mask
F02230	BLIT_A2PTR	RW	Blitter A2 pointer
F02234	BLIT_A2STEP	W	Blitter A2 step
F02238	BLIT_CMD	W	Blitter command
F0223C	BLIT_COUNT	W	Blitter loop counters
F02240	BLIT_SRCD	W	Blitter source data
F02248	BLIT_DSTD	W	Blitter destination data
F02250	BLIT_DSTZ	W	Blitter destination Z data
F02258	BLIT_SRCZ1	W	Blitter source Z data 1
F02260	BLIT_SRCZ2	W	Blitter source Z data 2
F02268	BLIT_PATD	W	Blitter pattern data
F02270	BLIT_IINC	W	Blitter intensity increment
F02274	BLIT_ZINC	W	Blitter Z increment
F02278	BLIT_STOP	W	Blitter collision stop control
F0227C	BLIT_I0	W	Blitter intensity register 0
F02280	BLIT_I1	W	Blitter intensity register 1
F02284	BLIT_I2	W	Blitter intensity register 2
F02288	BLIT_I3	W	Blitter intensity register 3
F0228C	BLIT_Z0	W	Blitter Z register 0
F02290	BLIT_Z1	W	Blitter Z register 1
F02294	BLIT_Z2	W	Blitter Z register 2
F02298	BLIT_Z3	W	Blitter Z register 3
F03000	GPU_RAMBASE	RW	Local RAM base

These locations may be accessed by all processors except the GPU for read or write as appropriate at the above addresses, where they appear to the system as 16-bit memory As they are all actually 32-bits, transfers should always be performed in pairs, in the order low address then high address

In addition, for high-speed write operations by 32-bit or 64-bit bus masters (especially for blit transfers), they may be written to as 32-bit locations at an offset of plus 8000 hex from the addresses above. They are not readable at these addresses

The GPU addresses them all directly as 32-bit locations in 32-bit internal memory, and they are not accessible to the GPU at the plus 8000 hex offset

Graphics Processor

This section describes the Jaguar Graphics Processor (GPU)

What is the Graphics Processor?

The Graphics Processor (called here the GPU - Graphics Processor Unit) is a simple, very fast, micro-processor It is intended for performing the functions associated with generating graphics, such as three-dimensional modelling, shading, fast animation, and unpacking compressed images

The graphics processor corresponds to the accepted notion of a RISC Processor (Reduced Instruction Set Computer) This means that

- most instructions execute in one tick
- all computational instructions involve registers
- memory transfers are performed by load/store instructions
- instructions are of a simple fixed format, with few addressing modes
- there is a wealth of registers, and local high-speed memory

It has several features to give high computational powers, including

- highly pipe-lined architecture
- one instruction per tick peak throughput
- internal program and data RAM
- register score-boarding
- sixty-four thirty-two bit registers
- ALU includes barrel shifter and parallel multiplier
- systolic matrix multiplication
- fast hardware divide unit
- high-speed interrupt response, including video object interrupts
- close coupling with the Blitter

Programming the Graphics Processor

The GPU is programmed in the same way as any other micro-processor It has a full instruction set with a broad range of arithmetic instructions, including add, subtract, multiply and divide, Boolean instructions, and bit-wise instructions It has a range of instructions for loading and storing values in memory, with either register indirect, register indirect plus register offset, or register indirect plus immediate offset addressing modes It has jump relative and absolute instructions, both of which may be made dependant on combinations of the zero, carry and negative flags There are also some more specialist instructions suited to computing matrix multiplies, and some useful aids to floating-point calculations

The GPU is a full 32-bit processor in that all internal data paths are 32-bits wide, and all arithmetic instructions (except multiply) perform 32-bit computations The instructions are 16-bits wide

The GPU has sixty-four internal 32-bit general purpose registers, of which thirty-two are visible at one time It also has lK of local high-speed 32-bit RAM, which is where its instructions and working data are normally stored It also has access to external memory via the 64-bit co-processor bus, and can perform byte, word, long-word and phrase data transfers on this bus It can also execute its instructions from external RAM

Design Philosophy

The GPU is a RISC processor, normally executing one instruction per tick, and therefore capable of very high instruction throughput. The RISC versus CISC debate is a complex one, and will not be discussed here. The RISC approach was chosen for the GPU principally because it occupies less silicon.

The RISC approach leads to a processor design without micro-code, effectively the instruction set is the micro-code, and most instructions execute in one tick. The advantage is that instructions are executed quicker, but the disadvantage is that some operations require more instructions to execute.

The GPU is also intended to perform rapid floating-point arithmetic. It has no floating-point instructions as such, but has some specific simple instructions that allow a limited precision floating-point library to be capable of in excess of I MegaFlop.

The GPU is intended to be programmed in assembly language, and not in a compiled language, as the tasks it is intended to perform are simple repetitive operations, best written in assembly language.

Pipe-Lining

The GPU design makes extensive use of pipe-lining to improve its throughput. This means that although the GPU can achieve a peak rate of one instruction per tick, each instruction is actually executed over several ticks, but only spends one tick at each pipe-line stage. It is important to understand this as it does have some significant consequences on GPU behaviour.

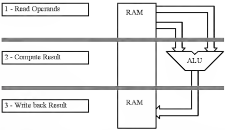
For a typical instruction, such as ADD, the pipe-line stages are:

1. decode instruction
2. read operands from registers
3. add operands
4. write result back to register

In addition to these stages, a pre-fetch unit attempts to maintain a small queue of unexecuted instructions, to keep the instruction execution unit busy.

Register Score-Boarding

The main side effect of the pipe-lined nature of GPU operation is the interaction of instructions at different stages of the pipe-line. They may affect the same operand, or the same piece of the hardware, and so a conflict can potentially arise.



For instance, if the instruction after an ADD was a second ADD of another value to the same register, then if the two instructions were just to follow each other through the pipe-line, then the second ADD would use the old value (the value from before the first ADD). Fortunately, the GPU hardware detects this erroneous condition and suspends execution until the correct value is ready. Clock cycles that occur during these hold-ups are referred to as *wait states*.

The figure shows the data flow associated with the operands of an arithmetic instruction. The thick lines correspond to a pipe-line stage, so that when an instruction is at the **Read Operands** stage, the previous instruction is at the **Compute Result** stage, and the one before that at the **Write Back Result** stage.

Two problems arise from this architecture:

1. The RAM used within the GPU for its registers has only two data ports, so if the instruction at stage three has to write back to a different register from the two registers being read by the instruction at stage one, then a clash occurs.
2. The instruction at stage one of the pipe-line may need to read a value being computed by the instruction at stage two, but this value will not be available until the instruction at stage two reaches stage three.

The GPU operates what is known as a *score-board* to help the programmer avoid a whole class of these problems. This tags registers that will alter once some operation has been completed, and will force program flow to wait if an instruction reads a tagged register. This mechanism also applies to the flags, and will wait if

- an instruction would read a register that is still in the process of being computed by the ALU.
- an instruction would perform a conditional jump, or add or subtract with carry, before the flags have been set as the result of some arithmetic operation.
- an instruction would read a register that is being read from internal memory.
- an instruction would read a register that is the target of a divide operation - as the divide unit is relatively slow, this can cause a significant delay.

- an instruction would read from a register that is waiting to be loaded from slow external memory (which takes a variable amount of time)

WARNING - No score-board protection applies to writes. Therefore, if two instructions both write to the same register and the first one completes after the second, the data will be written out of sequence. If they both write at the same time, then the results are unpredictable. This only appplies where the second instruction does not read the register.

Register Write-Back

The score-board unit also controls the writing back of computed values. The registers are a bank of dual-port RAM, so it is not possible to read two register values simultaneously while writing to a third.

If the register to be written back to is being read by the instruction currently at stage 1 of the pipe-line, or if one of the operands of that instruction does not involve a register read, then the write-back will be concealed. Otherwise, the instruction will be held up one cycle while the computed value is written back.

The score-board unit controls all operations that involve writing to registers, and will also generate a wait state if the instruction that would have executed reads two registers, neither of which is the target of the write. Write-back data sources are:

- the result of an ALU computation
- the result of a divide operation (this occurs in parallel with the ALU)
- the data from an internal load operation
- the data from an external load operation

If two of these are to be written back simultaneously, execution is always held up for a tick.

One technique that can be used to help avoid wait states from the score-board unit is to *interleave* two sets of calculations, i.e. ensure that consecutive instructions do not use the same registers, but that instructions two apart generally do.

See the warning above about write clashes.

Jump Instructions

Pipe-lining also affects the execution of jump instructions. The transfer of control does not occur until the instruction *after* the jump instruction has been executed. This can be confusing, but helps to increase the overall instruction throughput. The safest technique is to follow all jump instructions with a NOP (null operation), but it is quite reasonable to place almost any other instruction here - but see the notes below on program control flow.

Memory Interface

The Graphics Processor is intended to operate in parallel with the other processing elements in the Jaguar system. In order to do this, a well-behaved GPU program should only make occasional use of the main memory bus. The GPU therefore has four Kilobytes of local memory, organised as 1K locations of thirty-two bits.

This memory is intended to be used for both program and data. It can be cycled at the graphics processor clock rate, and so is extremely fast. It may be viewed as a simple cache RAM, with software cache control - this technique is known as *visible caching*. When the graphics processor is executing code out of internal RAM, program fetch cycles will occupy less than half the RAM bandwidth.

To load up a program into the RAM within the GPU, the best technique is to use the blitter. Set it to blit phrases, and use the 32-bit GPU address range (see below)

To the GPU programmer the local RAM, local hardware registers, and external memory all appear in the same address space. The GPU memory controller determines whether a transfer is local or external, and generates the appropriate cycle. The only difference to the programmer is that only 32-bit transfers are possible within the GPU local address space, whereas 8, 16, 32 or 64-bit transfers are permitted externally

The local RAM sits on an internal GPU 32-bit bus. Also present on this bus are various GPU control registers, and the Blitter control registers. When a GPU transfer occurs outside the local address space, a gateway connects the local bus to the main bus. If a sixty-four bit transfer is requested, a special register is used for the other half of the data

The address space is organised as follows

F02000 - F021FF	graphics processor control registers
F02200 - F022FF	Blitter registers
F02300 - F02FFF	reserved
F03000 - F03FFF	local RAM
F04000 - F0FFFF	reserved

This local address space is also available to external devices via the I/O mechanism

The GPU local bus can therefore perform transfers for three quite separate mechanisms. These are, in decreasing order of priority

- CPU I/O access
- Operand data transfer
- Instruction fetch

External View of GPU Space

The GPU internal address space is accessible by any other Jaguar bus master, i.e. the CPU, the Blitter and the DSP can all access GPU internal space. This is part of the Jaguar I/O space within Tom. This is normally viewed as 16-bit read/write memory, but by adding 8000 hex to the addresses it is also available as 32-bit write only memory, which is faster to access for a bus master which can perform 32-bit transfers. Specifically, this allows the blitter to copy data into the GPU space more rapidly than it would using the 16-bit space - for maximum transfer speed use the blitter in phrase mode, writing to the 32-bit address range.

The GPU and Data Ordering Conventions

The GPU can operate in both a big-endian and little-endian environment, and as long as the memory interface is programmed to the correct endian mode, and the transfer requested is the width of the operand required, then this operation is largely invisible to the programmer

The GPU instruction execution order may be little-endian or big-endian - with the exception that move immediate data is inherently little endian, i.e. it word ordering is least significant word then most significant word

Load and Store Operations

The GPU has a set of load and store instructions, each of which take two register operands. One register is used to provide the address, the other is either read to supply data to be stored or is written with load data.

Load and stores may be performed at byte, word, long-word and phrase width Bytes and words are aligned with bit 0, and when loaded the rest of the register is set to zero When phrases are read or written, a register within the GPU local address space should already contain the other long-word for store operations, or is loaded with the other long-word for load operations Performing phrase loads and stores is the fastest way of transferring blocks

Load and store operations may also be performed using one of two simple indexed addressing schemes These are both based on using either R14 or R15 as a base register, with either a five bit unsigned offset (in long words) encoded into one of the register fields or another register containing the offset There is a two tick overhead involved in using these instructions, as the address has to computed

In local memory, only long-word reads and writes are permitted.

Load and store operations will normally complete in one tick, or two ticks for indexed addresses The transfer may not be complete at this point, and if another load or store operation occurs before the previous one has completed it will be held up Load data is written under the control of the score-board unit, which is described elsewhere

The gateway between the GPU local bus and the external co-processor bus contains a control block for generating external memory transfers When this block is idle, load and store operations complete as quickly as they would in local memory For load operations, the data is not loaded into the target register, however, until the external transfer has taken place The score-board mechanism prevents use of this data before it has been loaded, but other computation may take place If there is another load or store instruction in the program before the gateway has completed its transfer, then it will be held up until the gateway is idle

Operand data transfers may occur at two bus priorities in external memory, either at the normal GPU priority, or at the higher DMA priority level This is controlled by the DMAEN flag This does not affect program reads, which are always at GPU priority Bus priority is discussed elsewhere This priority control bit must **not** be changed while an external memory cycle is active Note that these occur in the background, so be very careful about changing this flag dynamically, and do not modify it in an interrupt service routine

Note that it is quite safe to use the same register as both operands of a load (or store) operation These operations are quite legal

```
load  (r1),r1      ; over-write r1 with data after using it as address
load  (r14+2),r14  ; similarly, this is perfectly safe
store r2,(r2)      ; as is this, though less useful
```

Arithmetic Functions

The GPU contains a powerful ALU section, which as well as the normal arithmetic and Boolean functions, all with 32-bit word size, contains a 16 by 16 fast parallel multiplier, and a 32-bit barrel shifter, both of which perform their respective functions in one tick

The GPU also contains a divide unit This performs serial division at the rate of two bits per tick, on 32-bit unsigned operands, producing a 32-bit quotient The operation of this runs in parallel with normal GPU operation

The ALU has the following set of flags

Z	zero	set appropriately by all arithmetic operations, normally being set if the result of the operation was zero
N	negative	set appropriately by all arithmetic operations, normally being set if the result of the operation was negative (bit 31 is a one)

C	carry	set according to carry or borrow out of all add and subtract operations, set with the bit that is shifted out of shift and rotate operations for shift by one, left undefined by other arithmetic operations

Interrupts

The GPU can be interrupted by five sources. Interrupts force a call to an address in local RAM, given by sixteen times the interrupt number (in bytes), from the base of RAM. It is the responsibility of the programmer to preserve the registers and flags of the underlying code. Primary register 31 is the interrupt stack pointer. Primary register 30 is corrupted when instruction flow is transferred to the interrupt service routine. Neither register should be used for any other purpose when interrupts are enabled.

Interrupts are allocated as follows:

4	Blitter
3	Object Processor
2	Timing generator
1	DSP interrupt, the interrupt output from Jerry
0	CPU interrupt

The flags register contains individual interrupt enables for each of these sources, as well as a master interrupt mask for all interrupts. When the master interrupt mask is set, the primary register bank is selected (see below).

When an interrupt occurs, the master interrupt mask bit is set. The individual enables are not affected, but no other interrupts will be serviced until the mask bit is cleared. The interrupt service routine should normally clear the master interrupt mask, and the appropriate interrupt latch, and enable higher priority interrupts immediately.

The value pushed onto the R31 stack is the address of the last instruction to be executed before the interrupt occurred. The interrupt service routine should therefore add two to this value before using it to return from the interrupt.

The interrupt latches may be read in the status port, and are cleared by writing a one to their clear bits, writing a zero leaves them unchanged.

The cause of the interrupt may be determined by the location jumped to, but not from the flags register, as more than one interrupt latch bit may be set.

There is a certain degree of interrupt prioritization, in that if two interrupts arrive within a few ticks of each other, the higher numbered will be serviced first. Beyond this, interrupt prioritization is under software control, as described above.

The only operations that are atomic are single instructions, or certain instruction combinations (see below). Interrupts may be disabled by clearing all the enable bits. It is therefore not practical for the interrupt stack to be shared with the underlying code, unless all interrupts are masked across stack operations.

An example interrupt service routine, which does no more than clear the interrupt, is shown below. The interrupt source was interrupt 2.

```
int_serv:
      movei GPU_FLAGS,r30     ; point R30 at flags register
      load  (r30),r29         ; get flags
      bclr  3,r29             ; clear IMASK
      bset  11,r29            ; and interrupt 2 latch
      load  (r31),r28         ; get last instruction address
```

```
addq  2,r28           ; point at next to be executed
addq  4,r31           ; updating the stack pointer
jump  (r28)           ; and return
store r29,(r30)       ; restore flags
```

Similar interrupt service routines can handle all the interrupts. Note the following points about this code.

- Registers R28 and R29 may not be used by the under-lying code as they are corrupted, in addition to R30 and R31 which are always for interrupts only.
- Interrupts are re-enabled on the instruction after the jump. If they were enabled any sooner then no other interrupt service routine would be able to use R28 and R29, as they could potentially corrupt them before this service routine had completed,

If the interrupt source was the Object Processor, then the interrupt service routine should read the Object Code registers, if required, and then re-start the Object Processor by writing to the Object Processor Flag register, as quickly as possible.

Atomic Operations

It is necessary for certain operations to be atomic, i.e. interrupts may not occur during these operations. Three GPU instruction types temporarily lock out interrupts while they complete their operation. These are:

- Immediate data moves, using the MOVEI instruction. Interrupts are locked out while the two words of immediate data are fetched.
- Matrix multiply operations, using the MMULT instruction. Interrupts are locked out until the operation has completed.
- Multiply and accumulate operations, using the IMULTN and IMACN instructions. The result register is not preserved by interrupts, and therefore any multiply/accumulate operation must consist of a sequence of IMULTN and IMACN instructions followed by a RESMAC instruction, with no intervening instructions. The IMULTN and IMACN instructions are always atomic with the succeeding instruction. See the section below on multiply / accumulate instructions.
- Jump instructions are always atomic with the instruction which succeeds them.

Program Control Flow

Program control normally runs upwards through memory executing instructions sequentially. The GPU can also transfer program flow by performing jump instructions.

Two types of jump are supported, relative and absolute. Jump relative takes a signed five-bit offset, which is treated as an offset in words, and added to the program counter. Jump absolute transfers the contents of a register into the program counter.

Both types of jump may be conditional on the contents of the ALU flags. If the appropriate condition is not met, then the jump instruction is ignored and program flow continues with the next instruction after the jump.

The instruction after a jump is always executed. This is a side-effect of the pre-fetch queue. Programmers may choose either to place a NOP after every jump instruction, or may take advantage of this to place a useful instruction after the jump which will be executed whichever branch is followed.

The program counter may also be copied into a register.

The GPU can cease operation by clearing the GPUGO bit in the GPU control register (described below). It may then only be restarted by an external write to this register, or by a reset. Only the GPU can clear this bit, although any processor can set it (but the CPU can clear it when in single-stepping mode).

Single Step Operation

As an aid to the debugging of GPU programs, the GPU can be set to single step through programs, pausing between instructions until restarted. This operation is controlled by and external CPU as follows.

1- Set up the program counter, then set the GPUGO and SINGLE_STEP control bits in the control register.

2- Poll for the SINGLE_STOP flag in the status register - at this point the first instruction has been executed.

3- Set the SINGLE_GO bit in the control register (keeping GPUGO and SINGLE_STEP set).

4- Poll for the SINGLE_STOP flag being set (this is the read version of the SINGLE_STEP flag), which indicates that the next instruction has been executed.

5- Repeat from step 3.

If the GPU register file is to be read from or written to, then single-stepping will have to be suspended and an appropriate transfer routine run, which will require that the GPUGO bit must be cleared first and the program counter modified. Unfortunately, clearing the GPUGO bit has the effect of altering the value in the program counter, as the pre-fetch queue is discarded. Therefore, after step 4 above, the following operations should be performed:

- read the program counter value
- clear the GPUGO control bit
- read or write to the register file as required
- add two to the program counter value read
- restart from step 1 above

It is necessary to add two to the program counter, as the value read reflects the last instruction executed (or last word of immediate data if it was MOVEI).

Illegal Instruction Combinations

- Do not place a MOVEI instruction after a jump, as the jump will take effect before the data is fetched, and so will change where the immediate data is fetched from.
- Do not place two jump instructions sequentially, the results are not predictable, and may not be relied on.
- Do not place a MOVE PC to register instruction immediately after a jump absolute or jump relative instruction, the value read can not be relied upon.
- Do not follow an IMACN or IMULTN instruction by anything other than another than another IMACN instruction or a RESMAC instruction (see below).
- Do not precede an MMULT instruction by a LOAD or STORE instruction.

Conditional Jumps

Conditional jumps encode from a five bit flag field. This is

Bit	Condition
0	zero flag must be clear for jump to occur
1	zero flag must be set for jump to occur
2	flag selected by bit 4 must be clear for jump to occur
3	flag selected by bit 4 must be set for jump to occur
4	if set select negative flag, if clear select carry

This gives useful jumps as follows (other codes are either jump always or jump never, and are reserved for future modifications)

00000	0		Jump always
00001	1	NZ	Jump if zero flag is clear
00010	2	Z	Jump if zero flag is set
00100	4	NC	Jump if carry flag is clear
00101	5	NC NZ	Jump if carry flag is clear and zero flag is clear
00110	6	NC Z	Jump if carry flag is clear and zero flag is set
01000	8	C	Jump if carry flag is set
01001	9	C NZ	Jump if carry flag is set and zero flag is clear
01010	A	C Z	Jump if carry flag is set and zero flag is set
10100	14	NN	Jump if negative flag is clear
10101	15	NN NZ	Jump if negative flag is clear and zero flag is clear
10110	16	NN Z	Jump if negative flag is clear and zero flag is set
11000	18	N	Jump if negative flag is set
11001	19	N NZ	Jump if negative flag is set and zero flag is clear
11010	1A	N Z	Jump if negative flag is set and zero flag is set
11111	1F		Jump never

Multiply and Accumulate Instructions

The GPU supports multiply and accumulate (MAC) operations These involve multiplying two values together, and adding their product to the sum of the products of some previous multiply operations These are typically used for matrix multiply and digital filtering type applications

Due to the pipe-lined nature of the design, the multiply and its associated add do not take place in the same cycle MAC instructions are not therefore like other instructions, in that a special instruction is needed to write back their result

Take as an example multiplying R8 times R9, R10 times R11, R12 time R13, and placing the sum of their products in R2 All values are signed The instructions are as follows

```
imultn     r8,r9       ; compute the first product, into the result
imacn      r10,r11     ; second product, added to first
imacn      r12,r13     ; third product, accumulated in result
resmac     r2          ; sum of products is written to r2
```

MAC instructions may only be followed by further MAC instructions or by the RESMAC instruction No other combinations are permitted

Systolic Matrix Multiplies

The GPU contains a mechanism for performing integer matrix multiplies at a burst rate of the maximum obtainable from the hardware multiplier, which is one multiply per tick. This is generally useful, but has been designed in particular for the matrix multiplies required by the Discrete Cosine Transform algorithm. One technique for this involves performing two 8x8 integer matrix multiplies in succession on a matrix, using the same fixed coefficients, but rotated for the second multiply.

The GPU therefore has a MMULT instruction, which initiates a sequence of between three and fifteen multiply / accumulate instructions, as described above, corresponding to one product term of the result matrix. One of the source matrices is held in the secondary register bank, the other in local RAM. The matrix held in registers is packed, i.e. two elements per register. This allows all of an eight-by-eight matrix to be stored in the secondary register bank, and is the *raison d'être* of the second bank.

A matrix multiply is initiated by the MMULT instruction. This takes as its source parameter the register, which is always in the secondary register bank, containing the first two elements of the matrix row. Its destination parameter is the register, in the currently selected register bank, in which to write the result.

The matrix held in RAM may be accessed in either increasing row or increasing column order, in other words the data for each successive multiply operation are either one location or the matrix width apart.

Like interrupts, the systolic operation is performed by forcing internally generated instructions into the instruction stream. The first instruction is IMULTN, the middle ones IMACN, and the last RESMAC. These have their operands modified in the manner described above.

The MMULT instruction should not be preceded by a LOAD or STORE instruction.

Divide Unit

The divide unit performs unsigned division, taking as operands 32-bit divisor and dividend, giving a 32-bit quotient and a 32-bit remainder. The quotient is the result of the divide instruction, and replaces the dividend in the destination register. Divides are performed at the rate of two bits per tick, so that the complete divide operation completes in sixteen ticks. The divide instruction has no effect on the flags.

If another instruction attempts to read the quotient or start another divide operation while the divide unit is active, then wait states will be inserted until the divide unit has completed.

The remainder register may be read after the divide has completed, this value in this register may either be positive, in which case it contains the actual remainder, or negative, in which case it contains the remainder minus the divisor.

Divides may also be performed on unsigned 16.16 bit values, by setting the offset control flag in the divide control register. The quotient is then also an unsigned 16.16 bit value.

Register File

The GPU contains a register file of sixty-four thirty-two bit registers. All of them may be used as general purpose registers, although some are also assigned special functions.

All instructions contain two five-bit register operand fields, although they are not always used as such. Where an instruction references a register, this five-bit field is turned into the register address. There are two banks of these 32-bit registers, primary and secondary. The primary register bank, bank 0, is always used for interrupt

service This is forced by the IMASK bit, when it is set selection of bank 0 is forced If IMASK is clear REGPAGE is obeyed

Bank select bits are provided in the flags register, and special MOVE instructions allow data to be moved between banks

External CPU Access

The GPU internal address space is accessible to an external bus master at any time - external access having the highest priority on the GPU local bus This means that the Blitter may be used to load data into the local RAM

The local address space is accessible for read or write at the addresses given elsewhere in this document, and these locations are presented as sixteen bit memory, which must always be accessed as long words in the order low address then high address

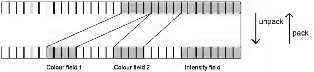
To allow faster transfers into the GPU space, all the registers are also available as thirty-two bit memory, at an offset of 8000 hex from their normal addresses At this address, the internal memory is write only

If the Blitter is being used to write into the GPU space, then phrase wide transfers may be performed, as the bus control mechanism will automatically divide these up to suit the width of the memory being addressed

Pack and Unpack

The **pack** and **unpack** instructions provide a means for averaging up to 32 CRY pixels The unpack operation leaves the intensity value unchanged, shifts the lower colour nibble up 5 bits, and the higher colour nibble up 10 bits The pack operation reverses this

Register containing packed pixel



Register containing unpacked pixel

There are five unused bits above each field in an unpacked pixel, allowing up to 32 unpacked pixels to be added together If a power of two unpacked pixel values are added, then a shift can be used to re-align them prior to packing the average value

The bits that do not contain packed or unpacked pixel data are always set to zero

This is useful for anti-aliasing and scaling effects

Instruction Set

The GPU instructions are all sixteen bits, made up as follows

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode						reg1					reg2				

- op code defines the instruction to be executed
- reg2 is the destination operand, or the only operand of single operand instructions
- reg1 is the source operand

The reg2 and reg1 fields usually hold a register number, but have other meanings with some instructions

The instruction set is as follows, where the syntax is

<Op code name> <source>,<destination>

Note The reg1 field of single operand instructions must always be set to zero for compatibility with manufacturing test modes and future enhancements

Flags

The description of each instruction indicates how it affects the flags The flags are valid when the result is written This is discussed further under "Writing Fast GPU Programs".

Register Usage

The description of register usage shows where it uses a register port Cycle 1 is the clock cycle at which the instruction is considered to be "executing", and is generally the pipe-line stage at which its register operans are read It is the only pipe-line stage occupied by NOP Where an instruction affects the flags, these are valid at the clock cyce when the result is written This is discussed further under "Writing Fast GPU Programs".

No	Syntax	Description
22	ABS Rn	**Absolute Value** 32-bit integer absolute value Has the same effect as NEG if the operand is negative, otherwise does nothing Note that this instruction does not work for value 8000000h, which is left unchanged, and with the negative flag set *Flags* Z - set if the result is zero N - cleared C - set if the operand was negative *Register Usage* Cycle 1 Destination register read Cycle 3 Destination register write

0	ADD Rn,Rn	**Add** 32-bit two's complement integer add, result is destination register contents added to the source register contents, and is written to the destination register *Flags* Z - set if the result is zero N - set if the result is negative C - represents carry out of the adder *Register Usage* Cycle 1 Source register read & Destination register read Cycle 3 Destination register write
1	ADDC Rn,Rn	**Add With Carry** 32-bit two's complement integer add with carry in according to the previous state of the carry flag, otherwise like ADD *Flags* Z - set if the result is zero N - set if the result is negative C - represents carry out of the adder *Register Usage* Cycle 1 Source register read & Destination register read Cycle 3 Destination register write
2	ADDQ n,Rn	**Add With Quick Data** 32-bit two's complement integer add, where the source field is immediate data in the range 1-32, otherwise like ADD *Flags* Z - set if the result is zero N - set if the result is negative C - represents carry out of the adder *Register Usage* Cycle 1 Destination register read Cycle 3 Destination register write
3	ADDQT n,Rn	**Add With Quick Data, Transparent** 32-bit two's complement integer add, like ADDQ except that it is transparent to the flags, which retain their previous values *Flags* ZNC - unaffected *Register Usage* Cycle 1 Destination register read Cycle 3 Destination register write
9	AND Rn,Rn	**Logical AND** 32-bit logical AND, the result is the Boolean AND of the source register contents and the destination register contents, and is written back to the destination register *Flags* Z - set if the result is zero N - set if the result is negative C - not defined *Register Usage* Cycle 1 Source register read & Destination register read Cycle 3 Destination register write

15	BCLR n,Rn	**Bit Clear** Clear the bit in the destination register selected by the immediate data in the source field, which is in the range 0-31. The other bits of the destination register are unaffected. *Flags* Z - set if destination register is now all zero N - set from bit 31 of the result C - not defined *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
14	BSET n,Rn	**Bit Set** Set the bit in the destination register selected by the immediate data in the source field, which is in the range 0-31. The other bits of the destination register are unaffected. *Flags* Z - set if the result is zero N - set if the result is negative C - not defined *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
13	BTST n,Rn	**Bit Test** Test the bit in the destination register selected by the immediate data in the source field, which is in the range 0-31. *Flags* Z - set if the selected bit is zero N - not defined C - not defined *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
30	CMP Rn,Rn	**Compare** 32-bit compare, this is the same as SUB without the result being stored, but the flags reflect the result of the comparison, which may therefore be used for equality testing and magnitude comparison. *Flags* Z - set if the result is zero (operands equal) N - set if the result is negative (source greater than destination operand) C - represents borrow out of the subtract *Register Usage* Cycle 1: Source register read & Destination register read Cycle 3: (flags are valid)

31	CMPQ n,Rn	**Compare With Quick Data** 32-bit compare with immediate data in the range -16 to +15 *Flags* Z - set if the result is zero (operands equal) N - set if the result is negative (immediate data greater than destination operand) C - represents borrow out of the subtract *Register Usage* Cycle 1: Destination register read Cycle 3: (flags are valid)
21	DIV Rn,Rn	**Unsigned Divide** The 32-bit unsigned integer dividend in the destination register is divided by the 32-bit unsigned integer divisor in the source register, yielding a 32-bit unsigned integer quotient as the result, like normal microprocessor division. The remainder is available, and division may also be performed on 16.16 bit unsigned integers. Refer to the section on arithmetic functions. *Flags* ZNC - unaffected *Register Usage* Cycle 1: Source register read & Destination register read Cycle 18: Destination register write
20	IMACN Rn,Rn	**Signed Integer Multiply/Accumulate, No Write-Back** 16-bit signed integer multiply and accumulate, like IMULT, except that the 32-bit product is added to the result of the previous arithmetic operation, and the result is not written back to the destination register. Intended to be used after IMULTN to give a multiply/accumulate group. * - refer to the section on Multiply and Accumulate instructions *Flags* ZNC - unaffected *Register Usage* Cycle 1: Source register read & Destination register read
17	IMULT Rn,Rn	**Signed Integer Multiply** 16-bit signed integer multiply, the 32-bit result is the signed integer product of the bottom 16-bits of each of the source and destination registers, and is written back to the destination register. *Flags* Z - set if the result is zero N - set if the result is negative C - not defined *Register Usage* Cycle 1: Source register read & Destination register read Cycle 3: Destination register write

18	IMULTN Rn,Rn	**Signed Integer Multiply, No Write-Back** Like IMULT, but result is not written back to destination register. Intended to be used as the first of a multiply/accumulate group, as there are potential speed advantages in not writing back the result. *Flags* Z - set if the result is zero N - set if the result is negative C - not defined *Register Usage* Cycle 1: Source register read & Destination register read
53	JR cc,n	**Jump Relative** Relative jump to the location given by the sum of the address of the next instruction and the immediate data in the source field, which is signed and therefore in the range +15 or -16 words. The condition codes encode in the same way as JUMP. *Flags* ZNC - unaffected *Register Usage* Cycle 1: (flags must be valid)
52	JUMP cc,(Rn)	**Jump Absolute** Jump to location pointed to by the source register, destination field is the condition code, where the bits encode as follows: Bit - Condition 0 - zero flag must be clear for jump to occur 1 - zero flag must be set for jump to occur 2 - flag selected by bit 4 must be clear for jump to occur 3 - flag selected by bit 4 must be set for jump to occur 4 - if set select negative flag, if clear select carry. If more than one condition is set, then they must all be true for the jump to occur (the conditions are ANDed). *Flags* ZNC - unaffected *Register Usage* Cycle 1: (flags must be valid)
41	LOAD (Rn),Rn	**Load Long** 32-bit memory read. The source register contains a 32-bit byte address, which must be long-word aligned. The destination register will have the data loaded into it. *Flags* ZNC - unaffected *Register Usage* Cycle 1: Source register read Cycle n: Destination register write (internal memory at cycle 3 or 4, external memory subject to bus latency)

43 44	LOAD (R14+n),Rn LOAD (R15+n),Rn	**Load Long, With Indexed Address** 32-bit memory read, as LOAD, except that the address is given by the sum of either R14 or R15 and the immediate data in the source register field, in the range 1-32. The offset is in long words, not in bytes, therefore a divide by four should be used on any label arithmetic to give the offset. This is slower than normal LOAD operations due to the two-tick overhead of computing the address. *Flags* ZNC - unaffected *Register Usage* Cycle 1: R14 or R15 register read Cycle n: Destination register write (internal memory at cycle 5 or 6, external memory subject to bus latency)
58 59	LOAD (R14+Rn),Rn LOAD (R15+Rn),Rn	**Load Long, From Register With Base Offset Address** 32-bit memory load from the byte address given by the sum of R14 and the source register (the address should be on a long-word boundary). Otherwise like instructions 43 and 44. *Flags* ZNC - unaffected *Register Usage* Cycle 1: R14 or R15 register read & Source register read Cycle n: Destination register write (internal memory at cycle 5 or 6, external memory subject to bus latency)
39	LOADB (Rn),Rn	**Load Byte** 8-bit memory read. The source register contains a 32-bit byte address. The destination register will have the byte loaded into bits 0-7, the remainder of the register is set to zero. This applies to external memory only, internal memory will perform a 32-bit read. *Flags* ZNC - unaffected *Register Usage* Cycle 1: Source register read Cycle n: Destination register write (external memory subject to bus latency)
40	LOADW (Rn),Rn	**Load Word** 16-bit memory read. The source register contains a 32-bit byte address, which must be word aligned. The destination register will have the word loaded into bits 0-15, the remainder of the register is set to zero. This applies to external memory only, internal memory will perform a 32-bit read. *Flags* ZNC - unaffected *Register Usage* Cycle 1: Source register read Cycle n: Destination register write (external memory subject to bus latency)

42	LOADP (Rn),Rn	**Load Phrase** 64-bit memory read The source register contains a 32-bit byte address, which must be phrase aligned The destination register will have the low long-word loaded into it, the high long-word is available in the high-half register This applies to external memory only, internal memory will perform a 32-bit read *Flags* ZNC - unaffected *Register Usage* Cycle 1 Source register read Cycle n Destination register write (external memory subject to bus latency)
54	MMULT Rn,Rn	**Matrix Multiply** Start systolic matrix element multiply, the source register is the location of the register source matrix, the product is written into the destination register Refer to the section on matrix multiplies The flags reflect the final multiply/accumulate operation *Flags* Z - set if the result is zero N - set if the result is negative C - represents carry out of the adder *Register Usage* Refer to the discussion of multiply/accumulate
34	MOVE Rn,Rn	**Move Register To Register** 32-bit register to register transfer *Flags* ZNC - unaffected *Register Usage* Cycle 1 Source register read Cycle 2 Destination register write
51	MOVE PC,Rn	**Move Program Count To Register** Load the destination register with the address of the current instruction The actual value read from the PC is modified to take into account the effects of pipe-lining and prefetch, to give the correct address This is the only way for the GPU to read its own PC *Flags* ZNC - unaffected *Register Usage* Cycle 2 Destination register write
37	MOVEFA Rn,Rn	**Move From Alternate Register** 32-bit alternate register to register transfer, the source register lying in the other bank of 32 registers *Flags* ZNC - unaffected *Register Usage* Cycle 1 Source register read Cycle 2 Destination register write

38	MOVEI n,Rn	**Move Immediate** 32-bit register load with next 32-bits of instruction stream The first word in the instruction stream is the low word, the second the high word *Flags* ZNC - unaffected *Register Usage* Cycle 3 Destination register write
35	MOVEQ n,Rn	**Move Quick Data** 32-bit register load with immediate value in the range 0-31 *Flags* ZNC - unaffected *Register Usage* Cycle 2 Destination register write
36	MOVETA Rn,Rn	**Move To Alternate Register** 32-bit register to alternate register transfer, the destination register lying in the other bank of 32 registers *Flags* ZNC - unaffected *Register Usage* Cycle 1 Source register read Cycle 2 Destination register write
55	MTOI Rn,Rn	**Mantissa To Integer** Extract the mantissa and sign from the IEEE 32-bit floating-point number in the source register, and create a signed integer in the destination The most significant bit is bit 23, but it is sign extended *Flags* Z - set if the result is zero N - set if the result is negative C - not defined *Register Usage* Cycle 1 Source register read Cycle 3 Destination register write
16	MULT Rn,Rn	**Multiply** 16-bit unsigned integer multiply, the 32-bit result is the unsigned integer product of the bottom 16-bits of each of the source and destination registers, and is written back to the destination register *Flags* Z - set if the result is zero N - set if bit 31 of the result is one C - not defined *Register Usage* Cycle 1 Source register read & Destination register read Cycle 3 Destination register write

8	NEG Rn	**Negate** 32-bit two's complement negate, the result is the destination register contents subtracted from zero, and is written back to the destination register Note that 80000000h cannot be negated *Flags* Z - set if the result is zero N - set if the result is negative C - represents borrow out of the subtract *Register Usage* Cycle 1 Source register read Cycle 3 Destination register write
57	NOP	**Do Nothing** *Flags* ZNC - unaffected *Register Usage* none
56	NORMI Rn,Rn	**Normalisation Integer** Gives the floating point normalisation integer for the value in the source register, which should be an unsigned integer The normalisation integer is the amount by which the source should be shifted right to normalise it as an IEEE 32-bit floating point value (the normalisation integer can be negative), and is also the amount to be added to the exponent to account for the normalisation *Flags* Z - set if the result is zero N - set if the result is negative C - not defined *Register Usage* Cycle 1 Source register read Cycle 3 Destination register write
12	NOT Rn	**Logical NOT** 32-bit logical invert, the result is the Boolean XOR of FFFFFFFF hex and the destination register contents, and is written back to the destination register *Flags* Z - set if the result is zero N - set if the result is negative C - not defined *Register Usage* Cycle 1 Destination register read Cycle 3 Destination register write

10	OR Rn,Rn	**Logical OR** 32-bit logical or operation, the result is the Boolean OR of the source register contents and the destination register contents, and is written back to the destination register. *Flags* Z - set if the result is zero N - set if the result is negative C - not defined *Register Usage* Cycle 1: Source register read & Destination register read Cycle 3: Destination register write
63	PACK Rn	**Pack CRY Pixel** Takes an unpacked pixel value and packs it into a 16-bit CRY pixel. Bits 22 to 25 are mapped onto bits 12 to 15, bits 13 to 16 are mapped onto bits 8 to 11, and bits 0 to 7 are mapped onto bits 0 to 7. The reg1 field should be set to zero to differentiate this from UNPACK. See the section on Pack and Unpack. *Flags* ZNC - unaffected *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
19	RESMAC Rn	**Multiply/Accumulate Result Write** Takes the current contents of the result register and writes them to the register indicated. Intended to be used as the final instruction of a multiply/accumulate group. * - refer to the section on Multiply and Accumulate instructions *Flags* ZNC - unaffected *Register Usage* Cycle 3: Destination register write
28	ROR Rn,Rn	**Rotate Right** 32-bit rotate right by the bottom 5 bits of the source register. Can be used for ROL functions by complementing the value. *Flags* Z - set if the result is zero N - set if the result is negative C - represents bit 31 of the un-shifted data *Register Usage* Cycle 1: Source register read & Destination register read Cycle 3: Destination register write
29	RORQ n,Rn	**Rotate Right By Immediate Count** Immediate data version of ROR. Shift count may be in the range 1-32. Z - set if the result is zero N - set if the result is negative C - represents bit 31 of the un-shifted data *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write

32	SAT8 Rn	**Saturate To Eight Bits** Saturate the 32-bit signed integer operand value to an 8-bit unsigned integer. If it is negative it is set to zero, if it is greater than 255 it is set to 255. This is useful for computed intensities and so on, to counteract the effect of rounding errors. *Flags* Z - set if the result is zero N - cleared C - not defined *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
33	SAT16 Rn	**Saturate To Sixteen Bits** Saturate the 32-bit signed integer operand value to a 16-bit unsigned integer. If it is negative it is set to zero, if it is greater than 65535 it is set to 65535. This is useful for computed Z, audio values, and so on, to counteract the effect of rounding errors. *Flags* Z - set if the result is zero N - cleared C - not defined *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
62	SAT24 Rn	**Saturate To Twenty Four Bits** Saturate the 32-bit signed integer operand value to a 24-bit unsigned integer. If it is negative it is set to zero, if it is greater than 16,777,215 it is set to 16,777,215. This is particularly useful for computed intensities, to counteract the effect of rounding errors. *Flags* Z - set if the result is zero N - cleared C - not defined *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
23	SH Rn,Rn	**Shift** 32-bit shift left or right given by the value in the source register. A positive value causes a shift to the right. Values of plus or minus thirty-two or greater give zero. Zero is shifted in. *Flags* Z - set if the result is zero N - set if the result is negative C - represents bit 0 of the un-shifted data for right shift, or bit 31 for left shift *Register Usage* Cycle 1: Source register read & Destination register read Cycle 3: Destination register write

26	SHA Rn,Rn	**Shift Arithmetic** As SH but right shift is arithmetic, i.e. sign shifted in. *Flags* Z - set if the result is zero N - set if the result is negative C - represents bit 0 of the un-shifted data for right shift, or bit 31 for left shift *Register Usage* Cycle 1: Source register read & Destination register read Cycle 3: Destination register write
27	SHARQ n,Rn	**Shift Arithmetic Right With Immediate Shift Count** As SHRQ but arithmetic shift right, i.e. sign shifted in. Best mnemonic. *Flags* Z - set if the result is zero N - set if the result is negative C - represents bit 0 of the un-shifted data *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
24	SHLQ n,Rn	**Shift Left With Immediate Shift Count** 32-bit shift left by n positions, in the range 1-32. Otherwise like SH (The shift value is actually encoded as 32-n, this is handled by the assembler) *Flags* Z - set if the result is zero N - set if the result is negative C - represents bit 31 of the un-shifted data *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
25	SHRQ n,Rn	**Shift Right With Immediate Shift Count** As SHLQ but shift right, zero shifted in *Flags* Z - set if the result is zero N - set if the result is negative C - represents bit 0 of the un-shifted data *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
47	STORE Rn,(Rn)	**Store Long** 32-bit memory write. The source register contains a 32-bit byte address, which must be long-word aligned. The destination register contains the data to be written. *Flags* ZNC - unaffected *Register Usage* Cycle 1: Source register read & Destination register read

49 50	STORE Rn,(R14+n) STORE Rn,(R15+n)	**Store Long, With Indexed Address** 32-bit memory write, write as STORE, with address generation in the same manner as the equivalent LOAD instructions *Flags* ZNC - unaffected *Register Usage* Cycle 1 R14 or R15 register read Cycle 2 Source register read
60 61	STORE Rn,(R14+Rn) STORE Rn,(R15+Rn)	**Store Long, To Register With Base Offset Address** 32-bit memory store to the byte address given by the sum of R14 and the destination register (the address should be on a long-word boundary) Otherwise like instructions 49 and 50 *Flags* ZNC - unaffected *Register Usage* Cycle 1 R14 or R15 register read & Destination register read Cycle 2 Source register read
45	STOREB Rn,(Rn)	**Store Byte** 8-bit memory write The source register contains a 32-bit byte address The destination register has the byte to be written in bits 0-7 This applies to external memory only, internal memory will perform a 32-bit write *Flags* ZNC - unaffected *Register Usage* Cycle 1 Source register read & Destination register read
48	STOREP Rn,(Rn)	**Store Phrase** 64-bit memory write The source register contains a 32-bit byte address, which must be phrase aligned The destination register contains the low long-word of the data to be written, the high long-word is obtained from the high-half register This applies to external memory only, internal memory will perform a 32-bit write *Flags* ZNC - unaffected *Register Usage* Cycle 1 Source register read & Destination register read
46	STOREW Rn,(Rn)	**Store Word** 16-bit memory write The source register contains a 32-bit byte address, which must be word aligned The destination register has the word to be written in bits 0-15 This applies to external memory only, internal memory will perform a 32-bit write *Flags* ZNC - unaffected *Register Usage* Cycle 1 Source register read & Destination register read

4	SUB Rn,Rn	**Subtract** 32-bit two's complement integer subtract, result is the source register contents subtracted from the destination register contents, and is written to the destination register The carry flag represents borrow out of the subtract, and the zero flag is set if the result is zero *Flags* Z - set if the result is zero N - set if the result is negative C - represents borrow out of the subtract *Register Usage* Cycle 1 Source register read & Destination register read Cycle 3 Destination register write
5	SUBC Rn,Rn	**Subtract With Borrow** 32-bit two's complement integer subtract with borrow in according to the carry flag, otherwise like SUB *Flags* Z - set if the result is zero N - set if the result is negative C - represents borrow out of the subtract *Register Usage* Cycle 1 Source register read & Destination register read Cycle 3 Destination register write
6	SUBQ n,Rn	**Subtract With Immediate Data** 32-bit two's complement integer subtract, where the source field is immediate data in the range 1-32, otherwise like SUB *Flags* Z - set if the result is zero N - set if the result is negative C - represents borrow out of the subtract *Register Usage* Cycle 1 Destination register read Cycle 3 Destination register write
7	SUBQT n,Rn	**Subtract With Immediate Data, Transparent** 32-bit two's complement integer subtract, like SUBQ except that it is transparent to the flags, which retain their previous values *Flags* ZNC - unaffected *Register Usage* Cycle 1 Destination register read Cycle 3 Destination register write

63	UNPACK Rn	**Unpack CRY Pixel** Takes an packed CRY pixel value and unpacks it into a 32-bit integer Bits 12 to 15 are mapped onto bits 22 to 25, bits 8 to 11 are mapped onto bits 13 to 16, and bits 0 to 7 are mapped onto bits 0 to 7 All other bits are set to zero The reg1 field should be set to one to differentiate this from PACK See the section on Pack and Unpack *Flags* ZNC - unaffected *Register Usage* Cycle 1 Destination register read Cycle 3 Destination register write
11	XOR Rn,Rn	**Logical XOR** 32-bit logical exclusive or, the result is the Boolean XOR of the source register contents and the destination register contents, and is written back to the destination register *Flags* Z - set if the result is zero N - set if the result is negative C - not defined *Register Usage* Cycle 1 Source register read & Destination register read Cycle 3 Destination register write

Internal Registers

This section describes the internal registers of the Graphics processor Note that some of these are read or write only

All GPU registers are 32-bit, and will require all 32 bits to be written

GPU Flags Register F02100 Read/Write

This register provides status and control bit for several important GPU functions Control bits are

0	ZERO_FLAG	The ALU zero flag, set if the result of the last arithmetic operation was zero Certain arithmetic instructions do not affect the flags, see above
1	CARRY_FLAG	The ALU carry flag, set or cleared by carry/borrow out of the adder/subtract, and reflects carry out of some shift operations, but it is not defined after other arithmetic operations
2	NEGA_FLAG	The ALU negative flag, set if the result of the last arithmetic operation was negative
3	IMASK	Interrupt mask, set by the interrupt control logic at the start of the service routine, and is cleared by the interrupt service routine writing a 0 Writing a 1 to this location has no effect

4-8	INT_ENA0-4	Interrupt enable bits for interrupts 0-4. The status of these bits is overridden by IMASK Interrupts are allocated as follows. 4 Blitter 3 Object Processor 2 Timing generator 1 DSP interrupt, the interrupt output from Jerry 0 CPU interrupt
9-13	INT_CLR0-4	Interrupt latch clear bits. These bits are used to clear the interrupt latches, which may be read from the status register. Writing a zero to any of these bits leaves it unchanged, and the read value is always zero
14	REGPAGE	Switches from register bank 0 to register bank 1. This function is overridden by the IMASK flag, which forces register bank 0 to be used
15	DMAEN	When DMAEN is set, GPU LOAD and STORE instructions perform external memory transfers at DMA priority, rather than GPU priority. This has no effect on program data fetches, which continue at GPU priority. This bit must **not** be changed while an external memory cycle is active. Note that these occur in the background, so be very careful about changing this flag dynamically, and do not modify it in an interrupt service routine.

WARNING - writing a value to the flag bits and making use of those flag bits in the following instruction will not work properly due to pipe-lining effects. If it is necessary to use flags set by a STORE instruction, then ensure that at least one other instruction lies between the STORE and the flags dependent instruction.

Matrix Control Register F02104 Write only

This register controls the function of the MMULT instruction. Control bits are:

0-3	MWIDTH	Matrix width, in the range 3 to 15
4	MADDW	When set, this control bit make the matrix held in memory be accessed down one column, as opposed to along one row.

Matrix Address Register F02108 Write only

This register determines where, in local RAM, the matrix held in memory is.

2-11	MTXADDR	Matrix address

Data Organisation Register F0210C Write only

This register controls the physical layout of pixel data and GPU I/O registers. If its current contents are unknown, the same data should be written to both the low and high 16-bits.

0	BIG_IO	When this bit is set, 32-bit registers in the CPU I/O space are big-endian, i.e. the more significant 16-bits appear at the lower address.
1	BIG_PIX	When this bit is set the pixel organisation is big-endian. See the discussion elsewhere in this document

2	BIG_INSTR	Normally, instructions are executed from a long-word in the order low word then high word When this bit is set the execution ordering is reversed, i e high word then low word However, move immediate data remains little-endian, i e the data must always be in the order low word then high word in the instruction stream

GPU Program Counter F02110 Read/Write

The GPU program counter may be written whenever the GPU is idle (GPUGO is clear) This is normally used by the CPU to govern where program execution will start when the GPUGO bit is set

The GPU program counter may be read at any time, and will give the address of the instruction currently being executed If the GPU reads it, this must be performed by the MOVE PC,Rn instruction, and not by performing a load from it

The GPU program counter must always be written to before setting the GPUGO control bit When the GPUGO bit is cleared, the program counter value will be corrupted, as at this point the pre-fetch queue is discarded

GPU Control/Status Register F02114 Read/Write

This register governs the interface between the CPU and the GPU

0	GPUGO	This bit stops and starts the GPU The CPU or GPU may write to this register at any time, however only the GPU should clear this bit (unless single-stepping is enabled)
1	CPUINT	Writing a 1 to this bit allows the GPU to interrupt the CPU There is no need for any acknowledge, and no need to clear the bit to zero Writing a zero has no effect A value of zero is always read
2	GPUINT0	Writing a 1 to this bit causes a GPU interrupt type 0 There is no need for any acknowledge, and no need to clear the bit to zero Writing a zero has no effect A value of zero is always read
3	SINGLE_STEP	When this bit is set GPU single-stepping is enabled This means that program execution will pause after each instruction, until a SINGLE_GO command is issued The read status of this flag, SINGLE_STOP, indicates whether the GPU has actually stopped, and should be polled before issuing a further single step command A one means the GPU is awaiting a SINGLE_GO command
4	SINGLE_GO	Writing a one to this bit advances program execution by one instruction when execution is paused in single-step mode Neither writing to this bit at any other time, nor writing a zero, will have any effect Zero is always read
5	unused	Write zero
6-10	INT_LAT0-4	Interrupt latches The status of these bits indicate which interrupt request latch is currently active, and the appropriate bit should be cleared by the interrupt service routine, using the INT_CLR bits in the flags register Writing to these bits has no effect

11	BUS_HOG	When the GPU is executing code out of external RAM it will normally give up the bus between program fetches. This behaviour should allow the CPU to continue to run at the same time. Setting this bit causes the GPU to attempt to hold on to the bus between program fetches, which improves its execution speed, at the expense of any lower priority device using the bus.
12-15	VERSION	These bits allow the GPU version code to be read. Current version codes are: 1 Pre-production test silicon 2 First production release Future variants of the GPU may contain additional features or enhancements, and this value allows software to remain compatible with all versions. It is intended that future versions will be a superset of this GPU.

High Data Register F02118 Read/Write

This 32-bit register provides the high part of GPU phrase reads and writes. It is physically a single register, and therefore a phrase read followed by a phrase write will write back the same high data unless this register is modified.

Divide unit remainder F0211C Read only

This 32-bit register contains a value from which the remainder after a division may be calculated. Refer to the section on the Divide Unit.

Divide unit Control F0211C Write only

0	DIV_OFFSET	If this bit is set, then the divide unit performs division of unsigned 16.16 bit numbers, otherwise 32-bit unsigned integer division is performed.

Writing Fast GPU Programs

To get the most out of the GPU, it is important to avoid **pipe-line stalls**. The GPU can execute one instruction per clock cycle in ideal circumstances, but it is very easy for code to be subject to so many stalls that it only achieves around half this figure. It will be worthwhile for programmers to tune the innermost loops of their code for maximum performance, and the rules given here should help do that. A well written GPU program can usually achieve an instruction throughput of around three-quarters of the peak figure.

Pipe-line stalls usually occur in the GPU either because an instruction would otherwise use some system resource, such as a register or a flag, which is not valid, or it would use a piece of hardware that is currently fully occupied, or active from an earlier operation, such as the external memory interface. This is because the GPU makes significant use of *pipe-lining* to improve performance.

The register bank is a source of stalls because it has only two read/write ports, so that two reads, a read and a write, or two writes can occur in any given clock cycle. If a result is being written at the same time as an instruction that requires two reads, then a stall will occur — unless the write register matches one of the two read registers, in which case the write occurs and the write data is provided as if the read was taking place. The instruction set list shows the register usage of all instructions.

Instructions dependant on the flags can also be subject to stalls, the flags are not valid until the clock cycle in which the result is written back, so that if a ADD instruction is followed by a JUMP then a one clock cycle stall will ensue, the JUMP executing in the clock cycle in which the result of the ADD is written back.

Pipe-line stalls are incurred when

- an instruction reads a register containing the result of the previous instruction, one clock cycle of wait is incurred until the previous operation completes
- an instruction uses the flags from the previous instruction, one clock cycle of wait is incurred until the previous operation completes
- an ALU result, memory load value or divide result has to be written back and neither register operand of the instruction about to be executed matches, one clock cycle of wait is incurred to let the data be written
- two values are to be written back at once, one clock cycle of wait is incurred (this is unusual)
- an instruction attempts to use the result of a divide instruction before it is ready. Wait states are inserted until the divide unit completes the divide, between one and sixteen wait states can be incurred.
- a divide instruction is about to be executed and the previous one has not completed, between one and sixteen wait states can be incurred
- an instruction reads a register which is awaiting data from an incomplete memory read, this will be no more than one clock cycle from internal memory, but can be several clock cycles from external memory
- a load or store instruction is about to be executed and the memory interface has not completed the transfer for the previous ones (one internal load/store or two external loads/stores can be pending without holding up instruction flow)
- after a store instruction with an indexed addressing mode (one clock cycle)
- after a jump or jr (three clock cycles if executing out of internal memory)
- if the next instruction has not been read, this will only occur when executing out of external memory
- during a matrix multiply if the CPU accesses GPU internal space

The most common cause of pipe-line stalls is using a register which was altered by the previous instruction. For example consider this code fragment:

```
1       add    r3,r0            ; add offset to X
2       shrq   1,r0             ; apply scaling factor
3       add    r0,r4            ; add to base
4       add    r5,r1            ; add offset to Y
5       shrq   1,r1             ; apply scaling factor
6       add    r1,r6            ; add to base
```

Stalls will be incurred after instructions 1, 2, 4 and 5. If the code were laid out like this:

```
1       add    r3,r0            ; add offset to X
2       add    r5,r1            ; add offset to Y
3       shrq   1,r0             ; apply scaling factor
4       shrq   1,r1             ; apply scaling factor
5       add    r0,r4            ; add to base
6       add    r1,r6            ; add to base
```

No stalls would occur. This is an example if *interleaving*, and this is a powerful technique for speeding up GPU code. It is well worth the performance enhancement - 6 clock cycles instead of 10 in this example - to

ensure that your code is laid out like this. Obviously there is a considerable overhead in thinking this out, but for loops that are executed many times it is well worth doing.

Blitter

This section describes the Jaguar Blitter

What is the Blitter?

Blitter is an abbreviation for *bit block processor*. It purpose is to process, by filling or copying, blocks of bits or pixels. These blocks may be one contiguous piece, or they may be sub-blocks (such as rectangles) within a larger pixel array.

The Blitter may also be seen as a hardware engine designed for painting and moving pixels as quickly as possible - it performs a variety of graphics operations at a rate limited largely by the memory access speed. It is used as an aid to the GPU, allowing a GPU program to process high-level graphics operations, whilst the Blitter, in parallel, performs the low-level repetitive pixel-by-pixel operations.

For example, the GPU might calculate the co-ordinates and gradients associated with a polygon, while the Blitter draws the strips of pixels. Alternatively, the GPU might be processing text with attributes, and computing font addresses and window positions, while the Blitter paints the characters.

The Blitter can perform a variety of operations on blocks of memory, including:

- simple memory copies
- copies and fills of rectangles within windows
- line-drawing
- image rotation and scaling
- single-scans of polygons fills
- Gouraud shading
- Z-buffering

The Blitter can operate on 1, 2, 4, 8, 16 or 32 bit packed pixels, with considerable flexibility with regard to the memory layout.

The *tour de force* of the Blitter is its ability to generate Gouraud shaded polygons, using Z-buffering, in sixteen bit pixel mode. A lot of the logic in the Blitter is devoted to its ability to create these pixels four at a time, and to write them at a rate limited only by the bus bandwidth, using the GPU to calculate the Z and intensity gradients and start and stop pixels on a line-by-line basis. This will give the system the ability to generate realistic animated 3D graphics.

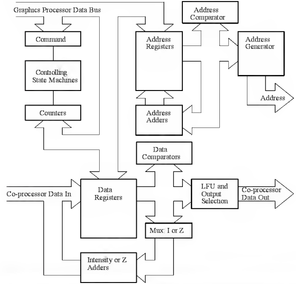
Programming the Blitter

The Blitter is programmed by setting up a description of the required operation in its registers. These are accessible in the system memory map, and so may be set by the GPU or by an external processor.

The registers control the three functional blocks that make up the Blitter, the address generator, data path, and control logic. Each of these is described in the sections that follow.

The descriptions that follow give a fairly dry account of how the Blitter works. These are useful for reference, but for an introduction to how to use the Blitter use the examples further on.

The Blitter architecture is summarised in the Figure below.



Address Generation

The address generator generates an address within a window of pixels. A window is a packed array of pixels in memory, and may well be the data associated with an Object Processor object. A window is described by its base address and width. A pointer into this window is set up for the Blitter start position, and is programmed in terms of its X and Y address. The ability to program the address generator in pixel address terms considerably simplifies the task of preparing Blitter commands.

In addition to these registers, various other registers contain specific values to allow considerable flexibility in how the pointers are modified during Blitter operations.

The Blitter has two address generation units, used for the *source* and *destination* addresses of copy operations, etc. The two address generators are called A1 and A2. A1 is normally the destination address register and A2 the source, although these roles may be reversed. A1 is more sophisticated in its address generation capabilities than A2.

The address register block looks like this:

F02200	A1 base address
F02204	A1 control flags
F02208	A1 clipping window size
F0220C	A1 pixel pointer
F02210	A1 step integer part
F02214	A1 step fractional part
F02218	A1 pixel pointer fractional part
F0221C	A1 increment integer part
F02220	A1 increment fractional part
F02224	A2 base address
F02228	A2 control flags
F0222C	A2 window address mask
F02230	A2 pixel pointer
F02234	A2 step integer part

Windows

All notions of address within the Blitter correspond with the concept of a window. A window is a rectangle of pixels, stored in memory as a linear array of packed phrases. A window is described by a base register, and has a width and height, both in pixels. A set of flags describe the size of those pixels, their physical layout in memory, and various aspects of how the pointer is updated.

The address itself is generated from a window pointer. This has an X and Y value, and again is in pixels. The pointer may point to areas outside the window, and A1 supports hardware clipping of addresses outside the window.

Address Generation

The X and Y pointers are sixteen bit values. However, the address generation mechanism will only generate valid addresses for Y values in the range 0-4095, i.e. it treats Y values as 12-bit unsigned values. The higher order bits of Y are ignored. X is treated as an unsigned 16-bit value, but only values from 0-32767 are valid in the blitter generally.

The address generator derives the window width from a very simple six-bit floating-point format. The width value has a four bit unsigned exponent, and a three bit mantissa, whose top bit is implicit, and which has the point after the implicit top bit. This is similar to a cut down version of the IEEE single precision format without the sign bit. It must give a whole number of phrases in the current pixel size. Valid exponent values are in the range 0-11.

For example, a window width of 640 is 1010000000 binary, i.e. 1.01 x 2^9. Therefore the mantissa takes the value 01 (implicit top bit), and the exponent 1001. The width is therefore 1001 01 in binary.

Note that there is a window bounds clipping mechanism for the A1 pointer, which treats the X and Y as signed sixteen bit values This is described elsewhere

Pointer Updating

Both Blitter address generators can update their pointers so that they describe a raster scan over a rectangle. Along a scan line, the pointer may be updated either by one pixel or to the next phrase boundary, depending on how the Blitter is currently operating. Refer to the Data Path section for further details.

At the end of a scan line, the pointer is updated by a step value, which is the distance in X and Y to the start of the next scan line This action of scan across the block, then step to the next start, is controlled by the Blitter's inner and outer control loops, the inner loop traversing a scan line, and the outer loop adding the step value. Thus the inner loop length is the block width, and the outer loop length the block height

In addition to these modes, both address registers have certain special modes.

A2 may have a Boolean mask applied to its pointer This is logically ANDed with the pointer, so that the pointers may not exceed the bounds of a rectangle, whose sides are a power of two pixels long. This is intended to repeat a source texture or pattern over a larger destination area, e.g. filling a wall with a repeated brick pattern

A1 supports address updates based on a Digital Differential Analyzer This technique produces successive address by adding an increment to the pointers, both of which have integer and fractional parts, and is used in particular for line-drawing and rotating images.

The pointer and increment of A1, in both X and Y, have sixteen bit integer parts and sixteen bit fractional parts. The step value used on the outer loop address update also has integer and fractional parts.

Data Path

The Blitter has a sixty-four bit data path, with a variety of registers It can be used to process entire phrases at once, or one pixel at a time Pixels may the one, two, four, eight, sixteen or thirty-two bits wide, and are always stored in a packed manner.

Data registers are

F02240	Source data, or computed intensity fractional parts
F02248	Destination data
F02250	Destination Z
F02258	Source Z1, or computed Z integer parts
F02260	Source Z2, or computed Z fractional parts
F02268	Pattern data, or computed intensity integer parts
F02270	Intensity increment
F02274	Z increment

When writing or copying pixels, arbitrary alignment of the source and destination data is allowed, and the Blitter aligns the source to match the destination data when required.

When transferring phrases the source and destination address pointers do not need to be aligned to the same point in a phrase, the Blitter will automatically align the source to the destination, but only for pixels of eight bits

or larger If two source phrase must be read before a destination phrase can be written, then the SRCENX flag must be set to ensure that enough source data is fetched for the blit to operate correctly

There are therefore two source data registers, to provide current source and previous source for alignment There is also a destination data register, which can be logically combined with the source, and is also used to restore the destination data area when only parts of it are updated

There is a parallel mechanism for Z data, used for Z-buffering This allows the depth of the data about to be written to be compared with the depth of the data already present on the screen, and the write of the new data inhibited if the data already present has a higher priority This applies to sixteen bit pixel mode only

There are therefore two source Z registers and a destination Z register

Write Data

Write data may come from

- the pattern data register
- the logic function unit
- computed Gouraud shaded data

The default is the LFU output The ADDDSEL flag selects adder output, PATDSEL selects the pattern register, and GOURD selects computed data

Write Z may come from

- source Z
- computed Z

The GOURZ flag selects computed Z data

Overriding both these selections is a mechanism to write back unchanged destination data If a mode is enabled where data may be inhibited, e g bit-to-byte expansion, or Z buffering, then a pre-read of the destination data should be performed This also applies to pixel sizes of less than eight bits

Data Comparators

There are three data comparators available within the Blitter These are

- The bit comparator This is used for bit to pixel expansion, and selects a bit or group of bits from the source data register, using a counter which is cleared every time the inner loop is entered The bit is then used to control whether a pixel is written at the current location
- The Z comparator This is used in 16-bit pixel mode to compare the 16-bit un-signed integer Z attribute of a pixel on the screen, the destination Z, with that about to be written, the source Z, and to prevent the write operation if the pixel on the screen has a higher priority
- The data comparator This is used to provide a means to make block copies with transparent colours, and to help with flood fill by performing searches It compares pixel values in either 8 or 16-bit pixel modes It normally compares the source data register with the pattern data registei, but it may also compare destination data with the pattern data

The comparators may be used to achieve three effects

- When painting pixels one at a time a comparator output can be used to inhibit the write of a pixel, leaving the previous value unchanged.

- When painting pixels a phrase at a time, the comparator outputs can force destination data to be written back. If this has been previously read then the data will be left unchanged, if not then a background colour can be used, stored in the destination data register
- The action of the Blitter can be stopped altogether This may be used for collision detection, searching, etc.

Note that the bit comparator can only produce a mask to operate over an entire phrase in 8-bit pixel mode

Bus Interface

The Blitter accesses memory through the 64-bit co-processor bus, and takes full advantage of the width and high-speed of this bus. The Blitter will normally cycle this bus at a rate limited only by the speed of the external memory, although there is a one-tick overhead when turning round from a read to a write transfer

All external memory is viewed by the Blitter as being phrase wide - if the physical layout is narrower then the memory controller expands the transfer into the appropriate number of transfers

The Blitter requests the bus at the start of an operation, and will not stop requesting it until the entire operation is complete. As described elsewhere, higher priority bus masters can request and be granted the bus during a Blitter operation, and this will suspend Blitter operation until the higher priority operation has released the bus.

Register Description

The following is a list of all the externally accessible locations within the Blitter The data registers may only be written to while the Blitter is idle

Address Registers

All address registers are 32-bits unless otherwise indicated The addresses given are byte offsets from the base of the GPU area

A1 Base Register F02200 Write only

32-bit register containing a pointer to the base of the window pointer to by A1 This address must be phrase aligned

Flags Register F02204 Write only

A set of flags controlling various aspects of the A1 window and how addresses are updated

Bits	Name	Description
0-1	Pitch	The distance between successive phrases of pixel data in the window data structure Gaps may be used to provide alternate pixel maps for double -buffering, for Z data, and for other control information The distance between two successive phrases of pixels is given by two to the power of this value, with one special case, i.e. a pitch of 0 means pixel data phrases are contiguous, 1 means 1 phrase gaps, 2 means 3 phrase gaps, but 3 means 2 phrase gaps, which may be especially useful for double -buffered Z-buffer displays, as it allows two phrases of pixels to each phrase of Z-buffer data - there is no need to double buffer the Z data
2	unused	
3-5	Pixel size	The pixel size, where the actual pixel size is 2^n, n is the value stored here Values 0-5 are allowed
6-8	Z offset	This value gives the offset from a phrase of pixel data of its corresponding Z data in phrases Values of 0 and 7 are not used
9-14	Width	This width is distinct from the width in pixels stored in the window register, and is the width used for address generation The width is a six-bit floating point value in pixels, with a four bit unsigned exponent, and a three bit mantissa, whose top bit is implicit, and which has the point after the implicit top bit This is similar to the IEEE single precision format without the sign bit It must give a whole number of phrases in the current pixel size For example, a screen width of 640 encodes as 1 01 x 2^9, where 1 01 is a binary number This gives an exponent field of 9, i.e 1001, and a mantissa field of (1)01 This is stored thus: Bit 14 13 12 11 10 9 E3 E2 E1 E0 M1 M0 1 0 0 1 0 1
15	unused	

16-17	X add ctrl	These control the update of the X pointer on each pass round the inner loop Values are: 00 - Add phrase width and truncate to phrase boundary (sets phrase mode) 01 - Add pixel size, effectively add one, 10 - Add zero 11 - Add the increment
18	Y add ctrl	This bit controls how the Y pointer is updated within the inner loop It is overridden by the X control bits if they are in add increment mode 0 - Add zero 1 - Add one
19	X sign	This bit may be set in conjunction with the X add pixel size mode to make the operation subtract pixel size It should not be set with other modes
20	Y sign	Makes the Y add one mode into Y subtract one

A1 Clipping Window Size F02208 Write only

This register contains the size in pixels, and may be used for clipping writes, so that if the pointer leaves the window bounds no write is performed The width is an unsigned fifteen bit value in the low word, the height an unsigned fifteen bit value in the high word The top bit of each word is ignored

The window origin (0,0) is always at the top left hand corner of the window, and so clipping is performed when the pointer values are negative, or when the pointer values are greater than or equal to these values If the desired clip rectangle does not have its top left corner at the window origin, then the window base register should be modified to make it the top left corner of the clip rectangle

A1 Window Pixel Pointer F0220C Read/Write

This register contains the X (low word) and Y (high word) pointers onto the window, and are the location where the next pixel will be written They are sixteen-bit signed values If X and Y values go out of range positively then they will advance through memory (X will wrap onto the next line, Y will go off the end of the window) Only X values in the range 0-32767 and Y values in the range 0-4095 will produce valid addresses from the address generator, values outside this range are for clipping purposes only

A1 Step Value F02210 Write only

The step register contains two signed sixteen bit values, which are the X step (low word) and Y step (high word) These may be added to the X and Y pointer on each pass round the outer loop, between passes through the inner loop

When calculating the step value for phrase-mode blits, note that the X pointer will be left pointing at the start of the first phrase not written by the blit

A1 Step Fraction Value F02214 Write only

The step fraction register may be added to the fractional parts of the A1 pointer in the same manner as the step value This is used when A1 is being used to scan over the source of a scaled or rotated image

A1 Window Pixel Pointer Fraction — F02218 — Read/Write

This register contains the fractional parts of the pointer when A1 is being used to implement a DDA based address generator, for line-drawing, etc The X part is in the low word, and the Y part in the high word

A1 Pixel Pointer Increment — F0221C — Write only

The increment is added to the pointer value within the inner loop when the address update is in add increment mode This register contains the two 16 bit signed integer parts of the increment, the X part is in the low word, the Y part in the high word

A1 Pixel Pointer Increment Fraction — F02220 — Write only

This is the fractional parts of the increment described above

A2 Base Register — F02224 — Write only

32-bit register containing a pointer to the base of the window pointer to by A2. This address must be phrase aligned.

A2 Flags Register — F02228 — Write only

A set of flags controlling various aspects of the A2 window and how addresses are updated

Bits	Name	Description
0-1	Pitch	As A1
2	unused	
3-5	Pixel size	As A1
6-8	Z offset	As A1
9-14	Width	As A1
15	Mask	Enables Boolean AND masking of the A2 pointer by its window register
16-17	X add ctrl	These control the update of the X pointer on each pass round the inner loop Values are: 00 - Add phrase width (truncate to phrase boundary) 01 - Add pixel size (effectively add one) 10 - Add zero
18	Y add ctrl	This bit controls how the Y pointer is updated within the inner loop. 0 - Add zero 1 - Add one
19	X sign	This bit may be set in conjunction with the X add pixel size mode to make the operation subtract pixel size It should not be set with other modes
20	Y sign	Makes the Y add one mode into Y subtract one

A2 Window Mask — F0222C — Write only

This register is used as the window size only in the sense that it may be used to AND mask the pointer register when the Mask flag is set This causes the address to wrap within a rectangular area and may be used to give fill patterns

A2 Window Pointer F02230 Read/Write

This register contains the X (low word) and Y (high word) pointers onto the window, and are the location where the next pixel will be written They are sixteen-bit signed values If X and Y values go out of range positively then they will advance through memory (X will wrap onto the next line, Y will go off the end of the window) Only X values in the range 0-32767 and Y values in the range 0-4095 will produce valid addresses from the address generator, values outside this range are for clipping purposes only

A2 Step Value F02234 Write only

The step register contains two signed sixteen bit values, which are the X step (low word) and Y step (high word) These may be added to the X and Y pointer on each pass round the outer loop, between passes through the inner loop

When calculating the step value for phrase-mode blits, note that the X pointer will be left pointing at the start of the first phrase not written by the blit

Control Registers

Command Register F02238 Write only

This register describes the operation of the Blitter A write to this register initiates Blitter operation, so it should be written to last when setting up a Blitter command Control bits are

Bit	Name	Description
Bits 0-5 enable corresponding memory cycles within the inner loop Destination write cycles are always performed (subject to comparator control), but all other cycle types are optional		
0	SRCEN	Enables a source data read as part of the inner loop operation
1	SRCENZ	Enables a source Z read as part of the inner loop operation This bit is ignored unless SRCEN is set
2	SRCENX	Enables an "extra" source data read at the start of an inner loop operation This is necessary where data has to be re-aligned, and may also sometimes be of use in bit-to-pixel expansion If SRCENZ is set an extra Z read is also performed
3	DSTEN	Enables a destination data read as part of inner loop operation This must always be performed for pixels smaller than 8 bits, where part of the destination data write will need to restore the data that was previously there
4	DSTENZ	Enables a destination Z read as part of inner loop operation
5	DSTWRZ	Enables a destination Z write as part of inner loop operation
6	CLIP_A1	Enables clipping when the A1 pointer lies outside its window boundaries This has the effect of inhibiting destination writes within the inner loop, but Blitter operation will continue
7	NOGO	Diagnostic use only, prevents write to the command register starting the Blitter Set to zero
Bits 8-10 enable address updates within the outer loop These should only be enabled when required as there is a one-tick overhead per update		
8	UPDA1F	Add the fractional part of the A1 step value to the fractional part of the A1 pointer between inner loop operations in the outer loop

9	UPDA1	Add the A1 step value to the A1 pointer between inner loop operations in the outer loop
10	UPDA2	Add the A2 step value to the A2 pointer between inner loop operations in the outer loop
11	DSTA2	Reverses the normal roles of the address registers from A1 as destination and A2 as source to A2 as destination and A1 as source
12	GOURD	Enable Gouraud shaded data updates within inner loop, i.e. the intensity gradient fractional part, repeated four times, is added to the computed intensity fraction register (a.k.a. destination data), then the intensity gradient integer part is added with the carry from the previous add to the computed intensity value register (a.k.a. pattern data).
13	GOURZ	Enable polygon Z data updates within the inner loop, i.e. add Z fractions to the Z fraction register (source Z 2), then add with carry the Z integer part to the Z integers (source Z 1).
14	TOPBEN	Enable carry into the top byte of the intensity integers in Gouraud data updates (leave clear for CRY mode).
15	TOPNEN	Enable carry into the top nibble of the intensity integers in Gouraud data updates (leave clear for CRY mode).

Bits 16-17 select alternative write data - the default source is the Logic Function Unit, whose output is controlled by the LFUFUNC bits

16	PATDSEL	Select pattern data as the write data
17	ADDDSEL	Selects the sum of source and destination data as the write data. Note that the source data is a signed offset. Leave TOPBEN and TOPNEN clear and the source data gives three signed offsets for each of the CRY fields, and the intensity value will saturate. Set TOPBEN and TOPNEN and sixteen bit saturating adds are performed. This can be used to lighten and darken images. This only applies to 16-bit pixels.
18-20	ZMODE	These bits give the conditions under which the Z comparator generates an inhibit. Setting them all to zero disables the Z comparator. This can only operate in 16-bit per pixel mode. bit 0 - source less than destination bit 1 - source equal to destination bit 2 - source greater than destination
21-24	LFUFUNC	The bits control the data produced by the logic function unit. The output is the Boolean OR of the following minterms: bit 0 - NOT source AND NOT destination bit 1 - NOT source AND destination bit 2 - source AND NOT destination bit 3 - source AND destination
25	CMPDST	Make the pixel value comparator compare destination data with pattern data rather than source data with pattern data
26	BCOMPEN	Enable write inhibit on the output from the bit comparator. This works pixel by pixel in any size, but over whole phrases only on 8-bit pixels. When operating in pixel mode then the write does not occur unless BKGWREN is set, but in phrase mode destination data is always written when the comprartor determines that the pixel should not be written.

27	DCOMPEN	Enable write inhibit on the output from the data comparator. This only applies to 8-bit and 16-bit per pixel modes. When operating in pixel mode then the write does not occur unless BKGWREN is set, but in phrase mode destination data is always written when the comprator determines that the pixel should not be written
28	BKGWREN	When a write inhibit occurs, this flag enables the Blitter to still perform the write, but to write back destination data. Thus only applies to pixel mode, in phrase mode destination data is always written
29	BUSHI	When set the blitter accesses the bus at the higher of its two priorities. This allows the blitter to access the bus at a higher priority than the object processor, and may speed up operations that involve a lot of short blits such as polygon drawing. Setting BUSHI across long blits may disturb the screen
30	SRCSHADE	This bit uses the IINC register to modify the intensity of data read from the source address, and may be used to lighten or darken images. It may be used in conjunction with GOURZ, but not GOURD. The data read from the source is modified, so source data should be selected using the LFU as the write data. This is particularly intended for performing flat shading on texture mapped surfaces

Status Register F02238 Read only

0	IDLE	When set, the blitter is completely idle and its last bus transaction is completed
1	STOPPED	When set, the blitter is stopped in its collision detection mode - see the collision control register below
2	inner IDLE	Diagnostic only
3	inner SREADX	Diagnostic only
4	inner SZREADX	Diagnostic only
5	inner SREAD	Diagnostic only
6	inner SZREAD	Diagnostic only
7	inner DREAD	Diagnostic only
8	inner DZREAD	Diagnostic only
9	inner DWRITE	Diagnostic only
10	inner DZWRITE	Diagnostic only
11	outer IDLE	Diagnostic only
12	outer INNER	Diagnostic only
13	outer A1FUPDATE	Diagnostic only
14	outer A1UPDATE	Diagnostic only
15	outer A2UPDATE	Diagnostic only
16-31	inner count	Diagnostic only

Counters Register F0223C Write only

The low word is the number of iterations of the inner loop operation. This is a sixteen bit value which reloads the inner loop counter on each entry to the inner loop.

The high word is the number of iterations of the outer loop. This is a sixteen bit value which is loaded directly into the outer loop counter.

The counters both accept values in the range 1 to 65536 (encoded as 0).

Data Registers

All data registers are sixty-four bits, unless otherwise noted

Source Data Register F02240 Write only

The source data may be pre-loaded with data for bit-to-byte expansion The source data register also serves to hold the four sixteen bit fractional parts of intensity when computing Gouraud shaded intensity

Destination Data Register F02248 Write only

This 64-bit register holds the destination data - which may be either read in the inner loop to allow unmodified pixels to be written back correctly when in phrase-mode, or it may be used to give background or paper colours, if it is not read

Destination Z Register F02250 Write only

This 64-bit register holds the destination Z value, and may be used as the data register

Source Z Register 1 F02258 Write only

The source Z register 1 is also used to hold the four integer parts of computed Z

Source Z Register 2 F02260 Write only

The source Z register 2 is also used to hold the four fraction parts of computed Z

Pattern Data Register F02268 Write only

The pattern data register also serves to hold the computed intensity integer parts and their associated colours

Intensity Increment F02270 Write only

This thirty-two bit register holds the integer and fractional parts of the intensity increment used for Gouraud shading Note that the top eight bits will modify the colour value, and should therefore normally be left set to zero

Z Increment F02274 Write only

This thirty-two bit register holds the integer and fractional parts of the Z increment used for computed Z polygon drawing

Collision control	**F02278**	**Write only**

This registers allows the Blitter to be stopped when an inner loop write inhibit occurs Blitter stop will occur in painting in pixel-by-pixel mode (X add control is 1), BKGWREN is clear, and one of BCOMPEN, DCOMPEN or ZMODE0-2 is set, along with the matching condition.

The Blitter operation may at that point be resumed or aborted

0	RESUME	Writing a one to this bit when the Blitter has stopped under the above conditions will cause the Blitter to resume operations Writing a zero has no effect
1	ABORT	Writing a one to this bit when the Blitter has stopped under the above conditions will cause the Blitter to terminate the current operation and revert to its idle state. Writing a zero has no effect
2	STOPEN	Set this bit to enable Blitter collision stops Clear it to disable them

Intensity 0	**F0227C**	**Write only**
Intensity 1	**F02280**	**Write only**
Intensity 2	**F02284**	**Write only**
Intensity 3	**F02288**	**Write only**

These four registers provide an alternate view of the computed intensity integer parts (pattern data) and computed intensity fractional parts (source data) registers. They are a convenient way of updating the intensity values for Gouraud shading. Each register is a 24 bit value (8 16 bit number), with the top eight bits unused, that modifies the corresponding fields of the computed intensity integer and fractional part registers Note that the colour fields in the pattern data registers are unaffected by writes to these registers.

Z 0	**F0228C**	**Write only**
Z 1	**F02290**	**Write only**
Z 2	**F02294**	**Write only**
Z 3	**F02298**	**Write only**

These registers are analogous to the intensity registers, and are for Z buffer operation. They affect the corresponding parts of the computed Z integer (source Z1) and computed Z fraction (source Z2) registers. They are 32 bit values (16 16 bit numbers)

Modes of Operation

This section discusses some of the typical modes of operation of the Blitter It is by no means a complete guide to all possible modes, but will show how to do certain common operations This is the best way to learn how to use the Blitter

Throughout this section, flags in flags registers that are not mentioned should always be set to zero Registers that are not mentioned need not be set up

Block Moves

The simplest of all Blitter operations is a block move, copying one area of memory onto another The Blitter will perform this operation one phrase at a time, and it is therefore a very rapid way of transferring data

The source address of the data should be stored in the A2 base register, and the destination address in the A1 base register If these are not phrase aligned addresses then they should be rounded down to a phrase boundary, and the offset (in the pixel size set) from the phrase boundary written into the X pointer The Y pointer should be set to zero

The length of the block should be stored in the inner counter - the number represents the number of pixels, so the largest block that can be copied is 32767 pixels, where 32-bit pixels are set this is 128K For smaller blocks it is usually easier to work in bytes The outer counter should be set to one

The Blitter needs to be told how to update the pointers after each read and write cycle, so the add control bits are set to zero to indicate phrase mode in both address flags registers

Having set these, a command is stored in the command register, with the SRCEN bit set to enable source reads, and the LFUFUNC bits set to 1100 to select source data If the source is not phrase alogned, then the SRCENX bit must be set

Rectangle Moves

Rectangle moves are very like block moves, but use a two-dimensional data set rather than the one-dimension of a block operation This brings in various new concepts

A two-dimensional array of pixels is stored in memory as a linear array of phrases This will usually be the data field of a bit-mapped object The Blitter has to know the width of this *window* of pixels As an address in the window, in pixel terms, is given by the X pointer plus the width times the Y pointer, a multiply operation is necessary to compute the address To avoid the need for a hardware multiplier in the Blitter address generator, the width is rather strangely encoded

Blitter window width is expressed as a floating-point number The actual value has a four-bit exponent and a three-bit mantissa, whose top bit is implicit This allows Blitter window widths to be any value whose binary form has no more than three significant digits followed by some number of zeroes

As an example, here are how various window widths encode

Value	Binary	Floating-point	Encoded
20	000000010100	1 01 x 2^4	0100 01
80	000001010000	1 01 x 2^6	0110 01
128	000010000000	1 00 x 2^7	0111 00
640	001010000000	1 01 x 2^9	1001 01
3584	111000000000	1 11 x 2^11	1011 11

The largest width value allowed is the last value one in this table - the smallest width is one phrase in the current pixel size. The width must always be a whole number of phrases in the current pixel size.

Rectangles are blitted like a raster scan, i.e. a line of pixels is transferred, then the pointer advances one line and transfers the next scan line of the rectangle. This jump from the end of one line to the start of the next is given by the ***step*** value. If pixels are being transferred one at a time, then the step value for X is the window width minus the rectangle width. If pixels are being transferred one phrase at a time, then the X pointer is left pointing at the start of the next phrase **after** the end of the block, and so the step value should be reduced accordingly.

Clipping may be performed by the A1 address generator, and simply prevents writes occurring at addresses outside the window boundaries, i.e. X or Y either negative or grater than the window size. The window size is programmed in the A1 window size registers. This is not much faster than writing the clipped pixels, so if a large number of pixels are to be clipped then it is worth performing the clipping at a higher level.

Character Painting

Character painting is a particular example of a class of operations requiring ***bit to pixel expansion***. As well as character painting, this may include such things as background patterns, simple texture fills, etc.

When bit to pixel expansion is being performed, the source data is used as a bit mask. Bits are extracted from the source data and if they are set then the corresponding pixel is painted in the currently selected output data form, if the bit is clear then either the pixel is left unchanged, or a background colour is written.

This allows character painting to paint the characters only, leaving the background unchanged (if the destination data is read), or with another colour written to the 'paper' areas (pre-loaded into the destination data register which is not read in the inner loop).

Character painting can be performed one pixel at a time in all screen modes, and can also be performed one phrase at a time in eight and sixteen bit per pixel modes.

The bit selection counter is reset every time the inner loop is left, so bit packed data patterns may be up to eight pixels wide.

Image Rotation

The Blitter can rotate and scale images as a single operation.

Consider taking a rectangular image and rotating it into a window:

- The bounding rectangle of the rotated image is calculated in the destination window.
- This rectangle is then transformed into the source image co-ordinate system.
- A2 is used as the destination address register and performs a raster scan over the bounding rectangle, pixel-by-pixel. The width and height of the blit are given by the size of this bounding rectangle.
- A1 performs a scan over the source image, with the increment integer and fraction set up to describe a scan over the first line of the translated bounding rectangle. The step and fraction parts then translate it to the start of the next scan.
- Clipping is generated when A1 is outside the bounds of the source image, so that writes at A2 will only be enables when A1 lies within the bounds of the source image, clipping the rotated form correctly.

Consider as an example, a 12 pixel square image starting at (10,10) in a window. We would like to rotate this image clockwise by 30 degrees, make it larger by a factor of 1.3, and move it across by 30 pixels.

First it is necessary to transpose the square's co-ordinates into the target co-ordinate system. The basic program below shows how to do this:

```
100 deg30 = .523598775
110 PRINT "Co-ordinates? "
120 INPUT xi, yi
130 x = xi - 16
140 y = yi - 16
150 xs = (x * COS(deg30)) - (y * SIN(deg30))
160 ys = (x * SIN(deg30)) + (y * COS(deg30))
170 x = xs * 1.3
180 y = ys * 1.3
190 x = x + 46
200 y = y + 16
210 PRINT "Translated: ", INT(x + .5), INT(y + .5)
```

This translates the vertices of the square as follows:

```
(10,10) -> (43,5)
(21,10) -> (56,12)
(21,21) -> (48,25)
(10,21) -> (36,18)
```

The bounding box is therefore from X = 36 to 56, and Y = 5 to 25. The vertices of this are then translated back to the source co-ordinate system, as shown by another basic program:

```
100 degm30 = -.523598775
110 PRINT "Co-ordinates? "
120 INPUT xi, yi
130 x = xi - 46
140 y = yi - 16
150 x = x / 1.3
160 y = y / 1.3
170 xs = (x * COS(degm30)) - (y * SIN(degm30))
180 ys = (x * SIN(degm30)) + (y * COS(degm30))
190 x = xs + 16
200 y = ys + 16
210 PRINT "Reverse translated: ", INT(x + .5), INT(y + .5)
```

This translates the vertices of the bounding box as follows:

```
(36,5)  -> (5,13)
(56,5)  -> (18,5)
(56,25) -> (26,18)
(36,25) -> (13,26)
```

We then set up A1 as the ***source*** address register, making its window base the top left hand corner of the source image, and its window size the image size. The A1 pointer will traverse the translated bounding box.

Gouraud Shading and Z-Buffering

Gouraud shading is a simple technique for modelling lit curved surfaces, which are represented by a series of polygons. To make the surface appear curved, the intensity must vary smoothly, rather than being uniform over each polygon. Gouraud shading approximates to the appearance of the curved surface by computing the intensity at each vertex, using a vertex normal, and some suitable illumination model. The vertex intensity is

then linearly interpolated across the polygon edges, and the edge intensities are linearly interpolated across the polygon scan lines.

Gouraud shading is only an approximation to the appearance of the curved surface, and may appear unnatural where there are large intensity changes across single polygons. However, it is much more attractive than not graduating the shading at all. Better shading can be achieved with Phong shading, where the normals are interpolated, but this is much more computationally intensive, and is not feasible within the Blitter.

Z-buffering involves attaching a Z value attribute to each pixel, which corresponds to how far away it is from the observer. When pixels are drawn on the screen, their Z values can be compared with the Z of the pixels already there, and the existing data preserved if closer to the observer. Z-buffering therefore provides a simple means of achieving hidden surface removal.

The Blitter can perform Gouraud shading and Z-buffering in sixteen bit pixel mode only. Each blit creates one scan line of a polygon, with the graphics processor responsible for re-calculating the start, length and gradient parameters for each scan line. Four pixels and their associated Z values can be calculated as fast as the memory interface can write them out, so the bus rate is always the limiting factor.

To calculate the Z and intensity values, the Blitter contains registers which represent the Z and intensity with a sixteen bit integer and sixteen bit fractional part. The intensity integer also contains the colour value, so intensity is prevented from overflowing into the colour information. The TOPBEN and TOPNEN bits enable this overflow, if desired.

There are four of these thirty-two bit values for intensity, and four for Z, so that four pixels may be calculated in parallel. There are also thirty-two bit Z and intensity increment registers, which give the amount added to each pixel for each write.

At each pass round the inner loop, the sixteen-bit fractional part of the intensity increment is added to the fractional parts of the intensity values, held in the source data register. Then the eight-bit integer part of the intensity is added with carry out of the fractional add to the integer pixel values in the pattern data register. Carry is prevented from propagating from intensity to colour. A similar mechanism governs Z.

Both the intensity and the Z values ***saturate***. This means that if they reach their lowest or highest values they are clipped there, rather than wrapping round. For example, adding one to a Z value of FFFF hex will give FFFF, not the overflow result 0000.

To take an example, consider blitting an 18 pixel strip of Gouraud shaded Z-buffered pixels. The Blitter command registers would be programmed as follows (all other registers need not be written).

Address registers are set up as follows:

```
A1_BASE         0x01600000  The window base address
A1_PITCH        1           Pixel data and Z data alternate
A1_PSIZE        4           16-bit pixels
A1_ZOFFS        1           Z data is one phrase up from pixel data
A1_WIDTH        0x11        20-pixel window: 1.01 x 2^4 = 0100 01
A1_ADDC         0           Add one phrase to address
A1_WIN_X        20          Window width
A1_WIN_Y        5           Window height
A1_PTR_X        1           First pixel at address 0,1
A1_PTR_Y        0
```

Data registers are set up assuming the first pixel has an intensity of C7.2833, and a colour of 00. The intensity gradient is minus 15.9265. The values for the first four pixels have to be set up (the left-most is actually off the edge of the strip, so the intensity gradient is subtracted from it). Similarly, the Z of the first pixel is E7E7.E000, and the Z gradient is minus 1818.1FFF.

```
Pattern      00DC00C700B1009C  Intensity integer parts and colour data
Source       FEDCEAC7D6B1C29C  Intensity fractions
Source Z1    FFFFE7E7CFCFB7B7  Z integer parts
Source Z2    FFFFE000C001A002  Z fractional parts
I Inc        FFA9B66C    Intensity increment (four times minus 15.9265)
Z Inc        9F9F8004    Z increment (four times minus 1818.FFFF)
```

Control information is set up as follows

```
Inner count 18    Strip width
Outer count 1     Single pixel high strip
DSTEN       1     Read destination data, to restore if necessary
DSTENZ      1     Read destination Z, to compare with computed Z
DSTWRZ      1     Write destination Z, restoring or replacing
CLIP_A1     1     Clip within window
GOURD       1     Gouraud data computation enabled
GOURZ       1     Z buffer data computation enabled
PATDSEL     1     Write pattern data
ZMODE       3     Overwrite existing data if the new Z value is
                  greater than or equal to the existing Z value
```

The numbers here are pretty arbitrary, but they show the general idea

Jerry

Jerry is the companion chip to Tom in the Jaguar games console Jerry provides the following functions

- A second RISC processor (DSP) principally intended for sound synthesis
- Frequency dividers for clock synthesis
- Two programmable timers
- Stereo PWM DAC (requires few external components)
- Synchronous serial interface and baud rate generator (I^2S)
- Asynchronous serial interface and baud rate generator (ComLynx)
- Joystick interface decodes
- Six general purpose IO decodes
- Two DMA channels (by way of DSP interrupts)

Jerry occupies a 64K byte slot in Jaguar's address space It appears as a 16 bit port (as does all IO) The DSP however is a 32 bit processor so all transfers to the DSP are done in pairs

Frequency dividers

Jerry is responsible for the synthesis of three important clocks

Chroma clock	This is 4 43 MHz for PAL and 3 58 MHz for NTSC and should have a 50% duty cycle
Video clock	This is a multiple of the pixel clock (which is typically between 6 MHz and 12 MHz) and must be tied to the chroma clock in order to avoid the "wood grain effect" on TVs
Processor clock	This determines the speed of the memory interface, the graphics processor, the object processor and the digital sound processor This clock is divided by two to provide a clock for an external processor

Jerry allows two approaches to clock synthesis

The less expensive approach is to derive chroma and video clocks from a crystal which is a multiple of the chroma clock and to generate the processor clock from a separate oscillator This is relatively inflexible it allows only a few horizontal resolutions e g 320, 480 and 640 pixels

The more expensive approach is to use PLLs with external phase comparators and VCOs The video clock and processor clock frequencies are then effectively continuously variable This technique is essential for gen-locking where the video clock phase comparator compares external and internal sync pulses

Three registers control the clock logic in Jerry. The ratio between the video clock and the pixel clock is determined by TOM.

CLK1	Processor clock divider	F10010	WO

This register is only used if the processor clock is generated by PLL. This ten bit register determines the frequency ratio between the processor clock oscillator input (PCLKOSC) and the processor clock divider output (PCLKDIV). In PLL clock synthesis PCLKDIV is typically locked to CHRDIV so the processor clock frequency will be

(N + 1) * CHRDIV

where N is the value written to this register. This register is initialised to one on reset. The PCLKDIV output produces a pulse every N + 1 PCLKOSC cycles.

CLK2	Video clock divider	F10012	WO

This register is only used if the processor clock is generated by PLL. This ten bit register determines the frequency ratio between the video clock (VCLK) and the video clock divider output (VCLKDIV). As before in PLL clock synthesis VCLKDIV is typically locked to CHRDIV so the video clock frequency will be

(N + 1) * CHRDIV

where N is the value written to this register. This register is initialised to zero on reset. The VCLKDIV output produces a pulse every N + 1 VCLK cycles.

CLK3	Chroma clock divider	F10014	WO

This six bit register determines the frequency ratio between the chroma oscillator (CHRIN, CHROUT) and the chroma clock divider output (CHRDIV). The divider divides the chroma oscillator frequency by N + 1 where N is the value written to the register. The CHRDIV output has a 50% duty cycle. This register is initialised to 3Fh (divide by 64) on reset. The most significant bit of this register enables the chroma oscillator onto the VCLK pin. This bit is clear on reset (output disabled).

Where PLL synthesis is used this register is typically left as reset This provides the lowest reference frequency for generating PCLK and VCLK.

For non-PLL synthesis the chroma crystal is some small multiple of the chroma carrier and this frequency is used as the video clock This register is written with the appropriate number to generate the chroma frequency on the CHRDIV pin and bit 15 is set to enable the crystal frequency onto the VCLK pin

Programmable Timers

Jerry contains two identical timers Each consists of two sixteen bit dividers The first stage (loosely called the pre-scaler) divides the processor clock by N + 1 The second stage divides this frequency by M +1, where N and M are the values written to their associated registers It is therefore possible to achieve frequency division in the range four to four billion

The outputs of the second stages may be used to interrupt either of the digital sound processor or the external microprocessor

It is intended that timer one is used to generate the sample rate frequency for sound synthesis and that timer two is used to generate a music tempo frequency The timers may however be used for other purposes It should be noted that writing to the associated registers presets the counters so they could be used to provide programmable delays Also the registers are readable which can be used to measure time accurately This might be used in development to help profile code or to help measure the time between joystick events

There are four registers associated with the timers The read addresses are different to the write addresses

JPIT1	**Timer 1 Pre-scaler**	**F10000**	**WO**
		F10036	**RO**
JPIT3	**Timer 2 Pre-scaler**	**F10004**	**WO**
		F1003A	**RO**

The pre-scalers divide the processor clock by N + 1 where N is the 16 bit value written to them The pre-scalers are down counters which are

loaded when the register is written and when they reach zero. They are readable, this is really for chip test purposes, but they might be used by the DSP to measure short events with precision.

The output of pre-scaler 1 is used by the PWM DACs to generate pulses. If these DACs are to be used then the value written to PIT1 must take account of this (see section on PWM DACs).

JPIT2	**Timer 1 Divider**	**F10002**	**WO**
		F10038	**RO**
JPIT4	**Timer 2 Divider**	**F10002**	**WO**
		F1003C	**RO**

These dividers divide the output from the corresponding pre-scalers by N + 1 where N is the 16 bit value written to them. The dividers, like the pre-scalers, are down counters which are loaded when the register is written and when they reach zero.

When they reach zero they may interrupt either of the DSP or the CPU. These interrupts are independently maskable.

Interrupts

There are six interrupt sources which may interrupt the external microprocessor. The interrupt sources are as follows:

- External — A rising edge on the EINT[0] input to Jerry may cause an interrupt
- DSP — The DSP may generate an interrupt by writing to a port
- Timers — Both timers may generate interrupts
- Sync — The synchronous serial interface can generate interrupts as described below
- UART — The asynchronous serial interface can generate interrupts as described below

It is likely that only one or two interrupt sources would normally be directed at the microprocessor. Some of the above are mainly of relevance to the DSP in sound synthesis. The Interrupt control register enables, identifies and acknowledges CPU interrupts from the six different interrupt sources.

INT	**Interrupt Control Register**	**F10020**	**RW**

Bits 0,8	External

Bits 1,9	DSP
Bits 2,10	Timer One (sample rate)
Bits 3,11	Timer Two (tempo)
Bits 4,12	Asynchronous Serial Interface
Bits 5,13	Synchronous Serial Interface

Bits 0 to 5 enable the individual interrupt sources When read bits 0 to 5 indicate which interrupts are pending Bits 8 to 13 clear

pending interrupts from the corresponding interrupt source

Pulse Width Modulation DACs

This logic allows stereo 14 bit DACs to be realised with a few inexpensive external components. The system works by breaking the 14 bit values into two 7 bit parts. It then generates pulses on four outputs with widths proportional to the 7 bit numbers. These pulses, which are generated at up to 240 KHz, are then weighted in proportion to their significance (128:1) using resistors then integrated and filtered to provide a signal with audio bandwidth.

The pulses are generated at the frequency generated by timer one pre-scaler. Pulses may be between 1 and 129 processor clock cycles wide so the pre-scaler must divide by at least 130 in order to guarantee a return to zero. If the pre-scaler divides by more than 130 then the audio output level will begin to drop. The pre-scaler can therefore be used to fine tune the sample rate interrupt.

The stereo values supplied to the PWM DACs need not be computed at the pulse frequency, but at an integer fraction of it. This is achieved by programming timer one divider to divide the pulse rate from the pre-scaler by that integer. The sample rate interrupt service routine should transfer the new left and right values to the DACs and initiate the computation for the next samples. The DACs are double buffered so the interrupt latency need only be less than the sample time. In practice the sample rate should tuned to the external low pass filter's characteristics.

The DAC registers can be written by any processor but the DSP can write to them without consuming any external bus bandwidth. The registers are two's complement and reset to all zeroes. Only the most significant fourteen bits are used. The PWM mechanism does not start until timer one is programmed. After initialisation the DACs should be written to with values decreasing from 8000 to zero at sample rate. This will avoid a loud click on start up.

There are two registers. These are within the local address space of the DSP, and so may be accessed by the DSP without any external bus overhead. Other processors may access them at these addresses. All transfers to them should be 32-bit, but the registers themselves are only 16-bit.

DAC1	**Left DAC**	**F1A140**	**WO**
DAC2	**Right DAC**	**F1A144**	**WO**

14-bit DAC registers as described above

Synchronous Serial Interface

The synchronous serial interface consists of four wires

- Receive data

input

- Transmit data output
- Serial clock in/out
- Word strobe

in/out

The clock and word strobe pins are outputs if Jerry is generating the timing for the serial interface (master) and inputs if Jerry uses externally generated timing (slave).

The interface can work in two modes The first, called mode16, is compatible with I²S and has a sixteen bit word length The start of left and right words are marked by transitions in word strobe Interrupts are generated on the rising edge of word strobe. The second mode, called mode32, allows longer packets of data to be communicated. In this mode a rising edge on word strobe synchronises the system which continues to receive/transmit 32 bit words Interrupts are generated every 32 bits

Mode16

```
               __    __    __    __    __    _   __    __    __
Clock        __/  \__/  \__/  \__/  \__/  \__  \__/  \__/  \__/  \__/
                   ___________________________  _______
Strobe       ____/                                     \_______________
             _____ _____ _____ _____ _____ __  _ _____ _____ _____ ___
Data         __1__X__0__X__15_X__14_X__13_X__  _X__1__X__0__X__15_X___

              left data | right data                        | left data
```

Note

- The word strobe precedes the data by one bit
- The word strobe and transmit data are clocked by the negative edge of the clock to provide the maximum set-up and hold time in the receiver/slave
- Data and word strobe inputs are sampled on the rising edge of the clock
- The data is sent transmitted MSB first If the interval between word strobe transitions is greater than 16 bits the transmitter sends zeroes after the LSB and the receiver ignores them If the

interval is less than 16 bits the receiver sets the missing bits to zero

- The diagram is the same whether the timing is generated internally or externally but Jerry only produces word strobes 16 bits in length

Mode32

```
               __    __    __    __    __     _   __    __    __
Clock        __/  \__/  \__/  \__/  \__/  \__   \__/  \__/  \__/  \__/
                   ______________________________
Strobe       ____/      \_____\_____\_____\__   ________________________
             _____ _____ _____ _____ _____ __   _ _____ _____ _____ ___
Data         __1__X__0__X__31_X__30_X__29_X__   _X__1__X__0__X__31_X___
```

Note

- Only the rising edge of the word strobe is significant
- Outputs change on the falling edge of the clock, and inputs are latched on the rising edge
- 32 bit words continue to be received / transmitted until the next rising edge of word strobe

The synchronous serial interface is controlled by seven registers These are all within the local address space of the DSP, and so may be accessed by the DSP without any external bus overhead Other processors may access them at these addresses All transfers to them should be 32-bit, but the registers themselves are only 16-bit The addresses given are therefore a big-endian view of their position in the memory map

SCLK Serial Clock Frequency F1A150 WO

This eight bit register determines the frequency of the internally generated serial clock The frequency is given by

Serial Clock Frequency = System Clock Frequency / (2 * (N+1))

where N is the number written to this register

SMODE Serial Mode F1A154 WO

Bit 0	INTERNAL	When set this bit enables the serial clock and word strobe outputs
Bit 1	MODE	When set this bit selects MODE32
Bit 2	WSEN	This bit enables the generation of word strobe pulses. When set JERRY produces a word strobe output which is alternately high for 16 clock cycles and low for 16 clock cycles When cleared Jerry will not generate further high pulses This can be used by software to generate one word strobe at the start of a packet of long-words in MODE32
Bit 3	RISING	Enables interrupts on the rising edge of word strobe

Bit 4	FALLING	Enables interrupts on the falling edge of word s trobe
Bit 5	EVERYWORD	Enables interrupts on the MSB of every word transmitted or received

LTXD	**Left transmit data**	**F1A148**	**WO**
RTXD	**Right transmit data**	**F1A14C**	**WO**

These two sixteen bit registers hold data to be transmitted
In MODE16 the right data is transferred to a shift register following the rising edge of word strobe and the left data is transferred following the falling edge of word strobe
In MODE32 the left data (most significant) is transferred first after the rising edge of word strobe (and every 32 clocks later), the right data is transferred 16 clocks after the left data
In either mode the registers may only be updated when the previous contents have been transferred to the shift register

LRXD	**Left receive data**	**F1A148**	**RO**
RRXD	**Right receive data**	**F1A14C**	**RO**

These two sixteen bit registers hold received data
In MODE16 the right data is transferred from the shift register to the register following the falling edge of word strobe and the left data is transferred following the rising edge
In MODE32 the left data (most significant) is transferred from the receive shift register to the left register 16 clocks after the rising edge of word strobe (and every 32 clocks later) The right data is transferred 16 clocks after the left data

SSTAT	**Serial Status**	**F1A150**	**RO**

Bit 0	WS	This bit reflects the state of the Word Strobe pin in order for software to determine which data is being received Do not use this signal for reading input data. Read the interrupt control register instead
Bit 1	Left	In MODE32 it is not necessary for the Word Strobe to be toggled every 16 bits An internal counter keeps track and this bit may be used as an alternative to WS to determine which word is currently being transmitted or received

Asynchronous Serial Interface (ComLynx and Midi)

The asynchronous serial interface consists of two wires, UARTI, the receive data input and UARTO the transmit data output. This interface is primarily designed to support ComLynx but it can also be used for MIDI transmit and receive.

A prescaler register is used to allow programmable baud rates.

The data transmitter is double buffered, allowing a character to be written into the data register before the transmission of a previously written character is complete. The data receiver is also double buffered, a second character can be received on the UARTI pin before the previous character has been read from the data register.

Data is both transmitted and received in the format shown below:

```
 ___      ___   ___   ___   ___   ___   ___   ___   ___   ___   _________
    \    / 0 \ / 1 \ / 2 \ / 3 \ / 4 \ / 5 \ / 6 \ / 7 \ /   \ /
     \__/\___/\___/\___/\___/\___/\___/\___/\___/\___/
     One                                                   One
     Start|------------ 8 Data bits ------------|Parity Stop
     Bit                                         Bit    Bit
```

The parity can be ODD, EVEN or none. The polarity of both the output and the input can be programmed to be active high or low. The polarity shown is active low.

Two classes of interrupt can be generated by the asynchronous serial interface, namely receiver or transmitter interrupts. Each of these classes can be individually enabled. The table below summarises the interrupts in each class.

Receiver Interrupts:

- Parity Error
- Framing Error
- Overrun Error
- Receive Buffer Full

Transmitter Interrupts

- Transmit Buffer Empty

ASICLK Asynchronous Serial Interface Clock F10034 R/W

This sixteen bit register determines the baud rate at which the asynchronous serial interface works. The frequency generated is given by:

Clock Frequency = System Clock Frequency / (N+1)

where N is the number written to this register.

The frequency generated by this register is further divided by sixteen to give the baud rate.

ASICTRL Asynchronous Serial Control F10032 WO

Bit 0	ODD	Writing a 1 to this bit selects odd parity

Bit 1	PAREN	Parity enable When parity is disabled the value of the ODD bit is transmitted in the parity bit time
Bit 2	TXOPOL	Transmitter output polarity Setting this bit to a one causes the UARTO output to be active low
Bit 3	RXIPOL	Receiver input polarity Writing a one to this bit makes the UARTI into an inverting input
Bit 4	TINTEN	Enables transmitter interrupts Note that the asynchronous serial interface bit in the Interrupt Control Register also needs to be set to enable interrupts
Bit 5	RINTEN	Enables receiver interrupts As for TINTEN the asynchronous serial interface bit in the Interrupt Control Register must also be set
Bit 6	CLRERR	Clear Error Writing a one to this bit clears any parity, framing or overrun error condition
Bit 14	TXBRK	Transmit break Setting this bit causes a break level to be transmitted on the UARTO pin It forces the UARTO output active This may be high or low depending on the state of the TXOPOL bit

All unused bits are reserved and should be written 0

ASISTAT Asynchronous Serial Status F10032 RO

Bits 0-5		These bits reflect the state of the corresponding bits in the ASICTRL register
Bit 7	RBF	Receive buffer full When set this bit indicates that a character has been received and is available in the ASIDATA register
Bit 8	TBE	Transmit Buffer Empty
Bit 9	PE	Parity Error This bit indicates that a parity error occurred on a received character
Bit 10	FE	Framing Error A framing error is detected when a non zero character is received without a stop bit at the expected time
Bit 11	OE	Overrun Error An overrun error is detected when a character is received on the input before the last character was read from the ASIDATA register
Bit 13	SERIN	Serial Input This bit reflects the state of the UARTI pin Its sense can be inverted by setting the RXIPOL bit in the ASICTRL register
Bit 14	TXBRK	Transmit Break This bit reflects the state of the corresponding bit in the ASICTRL register
Bit 15	ERROR	Error This bit is logical OR of the PE, FE and OE bits This allows a single test for error conditions

All unused bits are reserved and may return any value

ASIDATA Asynchronous Serial Data F10030 R/W

When this register is read it returns the last character received in bits [0 7] and zero in bits [8 15] The act of reading this register clears the receive buffer full condition leaving the way clear for subsequent characters to be received

When the ASIDATA register is written bits [0..7] are transmitted from the UART0 pin Bits [8..15] are not used and should be written as zero

Joystick Interface

Jerry has four outputs which together control four external TTL ICs to provide the joystick interface There are two registers

JOY1 Joystick register F14000 RW

When read the joystick input buffers are enabled and the data reflects the state of the sixteen joystick inputs Output JOYL0 is asserted (active low) during the read

When written the low eight data bits are latched into the joystick output latch Output J0YL2 is asserted (active low) during the write. The most significant bit (15) is used to enable the joystick outputs. This bit is cleared (disabled) by reset. Output J0YL3 is the inverse of the value in bit 15

J0Y2 Button register F14002 RW

When read the button input buffer is enabled and the data reflects the state of the four button inputs Output J0YL1 is asserted (active low) during the read

There are two joystick connectors each of which is a 15 pin high density 'D' socket. The pinouts are as follows

PIN	J5	J6
1	JOY3	JOY4
2	JOY2	JOY5
3	JOY1	JOY6
4	JOY0	JOY7
5	PAD0X	PAD1X
6	BO/LP0	B2/LP1
7	5 VDC	5 VDC
8	NC	NC
9	GND	GND
10	B1	B3
11	J0Y11	J0Y15
12	JOY10	JOY14
13	JOY9	JOY13

14	JOY8	JOY12
15	PAD0Y	PAD1Y

The JOYx signals correspond to bit x on the joystick port All the joystick signals can be used as inputs Signals JOY0 to J0Y7 can also be used as outputs. The direction of these signals is determined by bit15 of the joystick output port 1f bit 15 is set JOY0 to JOY7 are outputs All joystick signals are pulled up with resistors Signals B0 to B3 are bits 0 to 3 on the button port The PADx signals are analogue inputs The LP signals are light-gun inputs, a high level on these inputs transfers the current horizontal and vertical counts to the light-pen registers

General Purpose IO Decodes

Jerry has six general purpose IO decode outputs which are asserted (active low) in the following address ranges.

GPI00	F14800-F14FFFh	CD-interface
GPI01	F15000-F15FFFh	DMA ACK
GPI02	F16000-F16FFFh	Cartridge
GPI03	F17000-F177FFh	
GPI04	F17800-F17BFFh	
GPI05	F17C00-F17FFFh	Paddle Interface

The term "General Purpose" is a misnomer because most of the outputs are reserved.

DSP

Introduction

The DSP is part of the Jerry chip in Jaguar, and is a variant of the GPU within Tom. It uses a very similar instruction set and programming model, but there are certain differences The DSP has full access to the system memory map as a bus master, and its internal memory may be accessed by the other bus masters within the Jaguar System

The DSP performs two roles within Jaguar, its primary function is sound synthesis and it may also be available for additional graphics processing

Sound synthesis may be the playback of sampled sound or algorithmic sound generation, or a mixture of the two. As the DSP is a fast general purpose processor it may be used for a broad range of synthesis techniques It contains several optimisations for sound processing when compared to the GPU, in particular higher precision multiply / accumulate operations, circular buffer management, audio wave tables in local ROM, additional local fast RAM, and audio output hardware within its internal address space

As many sound generation techniques will not require anything like the full power of the DSP, it may also be used as an additional graphics processor It has full access to the entire system address space, although its bus bandwidth is lower as it has a 16-bit interface to external memory It might well be used with sound synthesis occurring under an interrupt at sample rate, with the underlying code performing something like matrix multiplies for 3D object rotation

This section assumes an understanding of the GPU, and outlines the differences between the GPU and the DSP

Programming the DSP

Refer to the 'Programming the Graphics Processor' *section in the GPU description*

Design Philosophy

Refer to the 'Design Philosophy' *section on the GPU description*

Pipe-Lining

Refer to the 'Pipe-Lining' *section on the GPU description*

Memory Map

Refer to the 'Memory Interface' *section of the GPU description for a discussion of the basics of the DSP memory interface*

The DSP has 8K bytes of local fast RAM (twice as much as the GPU), and 2K bytes of wave tables to help with sound synthesis These are laid out as follows

F1A000 - F1A1FF	DSP control registers
F1B000 - F1CFFF	local RAM
F1D000 - F1DFFF	wave table ROM

Wave Table ROM

The wave table ROM contains eight 128 entry wave tables These are signed 16-bit values, and are sign-extended to 32-bits, so that the ROM appears to occupy 1K 32-bit locations Only the bottom 16 bits are significant

The waves available are as follows

F1D000	TRI	A triangle wave
F1D200	SINE	A full wave SINE
F1D400	AMSINE	An amplitude modulated SINE wave
F1D600	SINE12W	A sine wave and its second order harmonic
F1D800	CHIRP16	A chirp - this is a sine wave increasing in frequency
F1DA00	NTRI	A triangle wave with noise superimposed
F1DC00	DELTA	A spike
F1DE00	NOISE	White noise

Load and Store Operations

Refer to the 'Load and Store Operations' *section of the GPU description.*

Arithmetic Functions

Refer to the 'Arithmetic Functions' *section of the GPU description*

The DSP replaces the unsigned saturation functions of the GPU with two signed operations SAT16S takes a signed 32-bit operand and saturates it to a signed 16-bit value, i e if it is less than $FFFF8000 it becomes $FFFF8000 and if it is greater than $00007FFF it becomes $00007FFF SAT32S takes a signed 40-bit operand (see the section below entitled 'Extended Precision Multiply / Accumulates') and saturates it to a signed 32 bit value in a similar manner

Interrupts

Refer to the 'Interrupts' *section of the GPU for a general discussion of how DSP interrupts behave*

There are six interrupts sources within the DSP These are allocated as follows

5	External interrupt 1
4	External interrupt 0
3	Timer interrupt 1
2	Timer interrupt 0
1	I^2S interface interrupt
0	CPU interrupt

The external interrupts are inputs from additional Jaguar hardware outside the Tom & Jerry system. The timer interrupts are from Jerry's local programmable timers, the I²S interrupt is from the local synchronous serial interface, and the CPU interrupt is generated by any processor writing to the DSP control register.

Program Control Flow

Refer to the 'Program Control Flow' *section of the GPU description.*

Circular Buffer Management

As circular buffers are common in DSP algorithms, for sample-looping, FIFOs, and so on, there is hardware support for addressing circular buffers. These have to be 2^n words long, and aligned to a 2^n boundary, where n is any practical value.

The support takes the form of two variants of ADDQ and SUBQ, namely ADDQMOD and SUBQMOD. These allow pointers to be updated with the value wrapping in the form of counting modulo 2^n. This is controlled by the modulo register which is a mask on the result of these instructions. Where a bit is 1 in this register, the result of the ADDQMOD or SUBQMOD is unaffected by the instruction, where it is 0 the add may modify it. Normally the high bits of this register are set to one, and the low bits set to zero as appropriate.

Extended Precision Multiply / Accumulates

Refer to the 'Multiply and Accumulate Instructions' *and the* 'Systolic Matrix Multiplies' *sections of the GPU description for an introduction to and explanation of these instructions.*

When multiply and accumulate operations are performed, using the IMULTN, IMACN and RESMAC instructions, or the MMULT instruction, the accumulated result is actually calculated as a forty bit signed integer. The top eight bits are effectively overflow bits, but they are not normally visible to the programmer. However, the SAT32S instruction takes as its forty bit input the register operand as the low thirty-two bits and the eight overflow bits of the accumulator as its top eight bits, and saturates the forty bit signed integer to thirty two bits; i.e. if it is less than FF80000000 it becomes FF80000000 and if it is more than 007FFFFFFF it becomes 007FFFFFFF.

The SAT32S instruction should therefore only be applied to the result of a multiply / accumulate operation, and before any further multiply / accumulate operations are performed. The SAT16S instruction operates only on its thirty-two bit register operand and takes no account of the overflow bits.

Divide Unit

Refer to the 'Divide Unit' *section of the GPU description.*

Register File

Refer to the 'Register File' *section of the GPU description.*

External CPU Access

Refer to the 'External CPU Access' *section of the GPU description.*

Addresses in DSP space are only available as 16-bit memory into which 32-bit transfers must be performed in the order low address then high address

Instruction Set

The DSP instructions are all sixteen bits, made up as follows

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode						reg1					reg2				

- op code defines the instruction to be executed
- reg2 is the destination operand, or the only operand of single operand instructions
- reg1 is the source operand

The reg2 and reg1 fields usually hold a register number, but have other meanings with some instructions.

The instruction set is as follows, where the syntax is

<Op code name> <source>,<destination>

Differences from the GPU Instruction set

- LOADP, SAT8, SAT16, SAT24, STOREP, PACK and UNPACK are absent
- SAT16S, SAT32S, ADDQMOD, SUBQMOD and MIRROR have been added

Nota Bene The reg1 field of single operand instructions must always be set to zero for compatibility with manufacturing test modes and future enhancements

No	Syntax	Description
22	ABS Rn	Absolute value 32-bit integer absolute value Has the same effect as NEG if the operand is negative, otherwise does nothing Note that this instruction does not work for value 8000000h, which is left unchanged, and with the negative flag set Z - set if the result is zero N - cleared C - set if the operand was negative
0	ADD Rn,Rn	Add 32-bit two's complement integer add, result is destination register contents added to the source register contents, and is written to the destination register Z - set if the result is zero N - set if the result is negative C - represents carry out of the adder
1	ADDC Rn,Rn	Add with carry 32-bit two's complement integer add with carry in according to the previous state of the carry flag, otherwise like ADD Z - set if the result is zero N - set if the result is negative C - represents carry out of the adder

2	ADDQ n,Rn	Add with quick data 32-bit two's complement integer add, where the source field is immediate data in the range 1-32, otherwise like ADD Z - set if the result is zero N - set if the result is negative C - represents carry out of the adder
63	ADDQMOD n,Rn	Add with quick data using modulo arithmetic 32-bit two's complement integer add like ADDQ, except that the result bits may be unmodified data if the corresponding modulo register bits are set This allows circular buffer management (for 2^n size buffers), where the high bits of the modulo register are set, and the low bits left clear Z - set if the result is zero N - set if the result is negative C - represents carry out of the adder
3	ADDQT n,Rn	Add with quick data, transparent 32-bit two's complement integer add, like ADDQ except that it is transparent to the flags, which retain their previous values ZNC - unaffected
9	AND Rn,Rn	Logical AND 32-bit logical AND, the result is the Boolean AND of the souce register contents and the destination register contents, and is written back to the destination register Z - set if the result is zero N - set if the result is negative C - not defined
15	BCLR n,Rn	Bit clear Clear the bit in the destination register selected by the immediate data in the source field, which is in the range 0-31 The other bits of the destination register are unaffected Z - set if destination register is now all zero N - set from bit 31 of the result C - not defined
14	BSET n,Rn	Bit set Set the bit in the destination register selected by the immediate data in the source field, which is in the range 0-31 The other bits of the destination register are unaffected Z - set if the result is zero N - set if the result is negative C - not defined
13	BTST n,Rn	Bit test Test the bit in the destination register selected by the immediate data in the source field, which is in the range 0-31 Z - set if the selected bit is zero N - not defined C - not defined

30	CMP Rn,Rn	Compare 32-bit compare, this is the same as SUB without the result being stored, but the flags reflect the result of the comparison, which may therefore be used for equality testing and magnitude comparison. Z - set if the result is zero (operands equal) N - set if the result is negative (source greater than destination operand) C - represents borrow out of the subtract
31	CMPQ n,Rn	Compare with quick data 32-bit compare with immediate data in the range -16 to +15 Z - set if the result is zero (operands equal) N - set if the result is negative (immediate data greater than destination operand) C - represents borrow out of the subtract
21	DIV Rn,Rn	Unsigned divide The 32-bit unsigned integer dividend in the destination register is divided by the 32-bit unsigned integer divisor in the source register, yielding a 32-bit unsigned integer quotient as the result, like normal microprocessor division. The remainder is available, and division may also be performed on 16.16 bit unsigned integers. Refer to the section on arithmetic functions. ZNC - unaffected
20	IMACN Rn,Rn	Signed integer multiply/accumulate, no write-back 16-bit signed integer multiply and accumulate, like IMULT, except that the 32-bit product is added to the result of the previous arithmetic operation, and the result is not written back to the destination register. Intended to be used after IMULTN to give a multiply/accumulate group. * - refer to the section on Multiply and Accumulate instructions ZNC - unaffected
17	IMULT Rn,Rn	Signed integer multiply 16-bit signed integer multiply, the 32-bit result is the signed integer product of the bottom 16-bits of each of the source and destination registers, and is written back to the destination register Z - set if the result is zero N - set if the result is negative C - not defined
18	IMULTN Rn,Rn	Signed integer multiply, no write-back Like IMULT, but result is not written back to destination register. Intended to be used as the first of a multiply/accumulate group, as there are potential speed advantages in not writing back the result. Z - set if the result is zero N - set if the result is negative C - not defined

53	JR cc,n	Jump relative Relative jump to the location given by the sum of the address of the next instruction and the immediate data in the source field, which is signed and therefore in the range +15 or -16 words. The condition codes encode in the same way as JUMP. ZNC - unaffected
52	JUMP cc,(Rn)	Jump absolute Jump to location pointed to by the source register, destination field is the condition code, where the bits encode as follows: Bit - Condition 0 - zero flag must be clear for jump to occur 1 - zero flag must be set for jump to occur 2 - flag selected by bit 4 must be clear for jump to occur 3 - flag selected by bit 4 must be set for jump to occur 4 - if set select negative flag, if clear select carry If more than one condition is set, then they must all be true for the jump to occur (the conditions are ANDed) ZNC - unaffected
41	LOAD (Rn),Rn	Load long 32-bit memory read. The source register contains a 32-bit byte address, which must be long-word aligned. The destination register will have the data loaded into it. ZNC - unaffected
43 44	LOAD (R14+n),Rn LOAD (R15+n),Rn	Load long, with indexed address 32-bit memory read, as LOAD, except that the address is given by the sum of either R14 or R15 and the immediate data in the source register field, in the range 1-32. The offset is in long words, not in bytes, therefore a divide by four should be used on any label arithmetic to give the offset. This is slower than normal LOAD operations due to the two-tick overhead of computing the address. ZNC - unaffected
58 59	LOAD (R14+Rn),Rn LOAD (R15+Rn),Rn	Load long, from register with base offset address 32-bit memory load from the byte address given by the sum of R14 and the source register (the address should be on a long-word boundary). Otherwise like instructions 43 and 44.
39	LOADB (Rn),Rn	Load byte 8-bit memory read. The source register contains a 32-bit byte address. The destination register will have the byte loaded into bits 0-7, the remainder of the register is set to zero. This applies to external memory only, internal memory will perform a 32-bit read. ZNC - unaffected
40	LOADW (Rn),Rn	Load word 16-bit memory read. The source register contains a 32-bit byte address, which must be word aligned. The destination register will have the word loaded into bits 0-15, the remainder of the register is set to zero. This applies to external memory only, internal memory will perform a 32-bit read. ZNC - unaffected

48	MIRROR Rn	Mirror operand The register is mirrored, i.e. bit 0 goes to bit 31, bit 1 to bit 30, bit 2 to bit 29 and so on. This is helpful for address generation in Fast Fourier Transform (FFT) operations. Z - set if the result is zero N - set if the result is negative C - not defined
54	MMULT Rn,Rn	Matrix multiply Start systolic matrix element multiply, the source register is the location of the register source matrix, the product is written into the destination register. Refer to the section on matrix multiplies. The flags reflect the final multiply/accumulate operation. Z - set if the result is zero N - set if the result is negative C - represents carry out of the adder
34	MOVE Rn,Rn	Move register to register 32-bit register to register transfer. ZNC - unaffected
51	MOVE PC,Rn	Move program count to register Load the destination register with the address of the current instruction. The actual value read from the PC is modified to take into account the effects of pipe-lining and prefetch, to give the correct address. This is the only way for the DSP to read its own PC. ZNC - unaffected
37	MOVEFA Rn,Rn	Move from alternate register 32-bit alternate register to register transfer, the source register lying in the other bank of 32 registers. ZNC - unaffected
38	MOVEI n,Rn	Move immediate 32-bit register load with next 32-bits of instruction stream. The first word in the instruction stream is the low word, the second the high word. ZNC - unaffected
35	MOVEQ n,Rn	Move quick data 32-bit register load with immediate value in the range 0-31. ZNC - unaffected
36	MOVETA Rn,Rn	Move to alternate register 32-bit register to alternate register transfer, the destination register lying in the other bank of 32 registers. ZNC - unaffected
55	MTOI Rn,Rn	Mantissa to integer Extract the mantissa and sign from the IEEE 32-bit floating-point number in the source register, and create a signed integer in the destination. The most significant bit is bit 23, but it is sign extended. Z - set if the result is zero N - set if the result is negative C - not defined

16	MULT Rn,Rn	Multiply 16-bit unsigned integer multiply, the 32-bit result is the unsigned integer product of the bottom 16-bits of each of the source and destination registers, and is written back to the destination register Z - set if the result is zero N - set if bit 31 of the result is one C - not defined
8	NEG Rn	Negate 32-bit two's complement negate, the result is the destination register contents subtracted from zero, and is written back to the destination register Note that 80000000h cannot be negated Z - set if the result is zero N - set if the result is negative C - represents borrow out of the subtract
57	NOP	Do nothing ZNC - unaffected
56	NORMI Rn,Rn	Normalisation integer Gives the 'normalisation integer' for the value in the source register, which should be an unsigned integer The normalisation integer is the amount by which the source should be shifted right to normalise it (the value can be negative), and is also the amount to be added to the exponent to account for the normalisation Z - set if the result is zero N - set if the result is negative C - not defined
12	NOT Rn	Logical NOT 32-bit logical invert, the result is the Boolean XOR of FFFFFFFF hex and the destination register contents, and is written back to the destination register Z - set if the result is zero N - set if the result is negative C - not defined
10	OR Rn,Rn	Logical OR 32-bit logical or operation, the result is the Boolean OR of the source register contents and the destination register contents, and is written back to the destination register Z - set if the result is zero N - set if the result is negative C - not defined
19	RESMAC Rn	Multiply/accumulate result write Takes the current contents of the result register and writes them to the register indicated Intended to be used as the final instruction of a multiply/accumulate group * - refer to the section on Multiply and Accumulate instructions ZNC - unaffected

28	ROR Rn,Rn	Rotate right 32-bit rotate right by the bottom 5 bits of the source register. Can be used for ROL functions by complementing the value Z - set if the result is zero N - set if the result is negative C - represents bit 31 of the un-shifted data
29	RORQ n,Rn	Rotate right by immediate count Immediate data version of ROR. Shift count may be in the range 1-32 Z - set if the result is zero N - set if the result is negative C - represents bit 31 of the un-shifted data
33	SAT16S Rn	Saturate to sixteen bits Saturate the 32-bit signed integer operand value to a 16-bit signed integer. If it is negative it is less than 8000h it is set to that, if it is greater than 7FFFh it is set to that Z - set if the result is zero N - cleared C - not defined
42	SAT32S Rn	Saturate multiply/accumulate result Saturate the 40-bit signed integer operand value to an 32-bit signed integer. This uses the overflow bits from multiply/accumulate operations as the top eight bits of the source value. If the accumulated value is less than 80000000h it saturates to that, if it is greater then 7FFFFFFFh it saturates to that Z - set if the result is zero N - set if the result is negative C - not defined
23	SH Rn,Rn	Shift 32-bit shift left or right given by the value in the source register. A positive value causes a shift to the right. Values of plus or minus thirty-two or greater give zero. Zero is shifted in. Z - set if the result is zero N - set if the result is negative C - represents bit 0 of the un-shifted data for right shift, or bit 31 for left shift
26	SHA Rn,Rn	Shift arithmetic As SH but right shift is arithmetic, i.e. sign shifted in. Z - set if the result is zero N - set if the result is negative C - represents bit 0 of the un-shifted data for right shift, or bit 31 for left shift
27	SHARQ n,Rn	As SHRQ but arithmetic shift right, i.e. sign shifted in. Best mnemonic Z - set if the result is zero N - set if the result is negative C - represents bit 0 of the un-shifted data

24	SHLQ n,Rn	Shift left with immediate shift count 32-bit shift left by n positions, in the range 1-32. Otherwise like SH. (The shift value is actually encoded as 32-n, this is handled by the assembler) Z - set if the result is zero N - set if the result is negative C - represents bit 31 of the un-shifted data
25	SHRQ n,Rn	Shift right with immediate shift count As SHLQ but shift right, zero shifted in Z - set if the result is zero N - set if the result is negative C - represents bit 0 of the un-shifted data
47	STORE Rn,(Rn)	Store long 32-bit memory write. The source register contains a 32-bit byte address, which must be long-word aligned. The destination register contains the data to be written ZNC - unaffected
49 50	STORE Rn,(R14+n) STORE Rn,(R15+n)	Store long, with indexed address 32-bit memory write, write as STORE, with address generation in the same manner as the equivalent LOAD instructions ZNC - unaffected
60 61	STORE Rn,(R14+Rn) STORE Rn,(R15+Rn)	Store long, to register with base offset address 32-bit memory store to the byte address given by the sum of R14 and the destination register (the address should be on a long-word boundary). Otherwise like instructions 49 and 50.
45	STOREB Rn,(Rn)	Store byte 8-bit memory write. The source register contains a 32-bit byte address. The destination register has the byte to be written in bits 0-7. This applies to external memory only, internal memory will perform a 32-bit write ZNC - unaffected
46	STOREW Rn,(Rn)	Store word 16-bit memory write. The source register contains a 32-bit byte address, which must be word aligned. The destination register has the word to be written in bits 0-15. This applies to external memory only, internal memory will perform a 32-bit write. ZNC - unaffected
4	SUB Rn,Rn	Subtract 32-bit two's complement integer subtract, result is the source register contents subtracted from the destination register contents, and is written to the destination register. The carry flag represents borrow out of the subtract, and the zero flag is set if the result is zero. Z - set if the result is zero N - set if the result is negative C - represents borrow out of the subtract

5	SUBC Rn,Rn	Subtract with borrow 32-bit two's complement integer subtract with borrow in according to the carry flag, otherwise like SUB Z - set if the result is zero N - set if the result is negative C - represents borrow out of the subtract
6	SUBQ n,Rn	Subtract with immediate data 32-bit two's complement integer subtract, where the source field is immediate data in the range 1-32, otherwise like SUB Z - set if the result is zero N - set if the result is negative C - represents borrow out of the subtract
32	SUBQMOD n,Rn	Subtract with immediate data 32-bit two's complement integer subtract like SUBQ, except that the result bits may be unmodified data if the corresponding modulo register bits are set This allows circular buffer management (for 2^n size buffers), where the high bits of the modulo register are set, and the low bits left clear Z - set if the result is zero N - set if the result is negative C - represents borrow out of the subtract prior to the modulo masking
7	SUBQT n,Rn	Subtract with immediate data, transparent 32-bit two's complement integer subtract, like SUBQ except that it is transparent to the flags, which retain their previous values ZNC - unaffected
11	XOR Rn,Rn	Logical XOR 32-bit logical exclusive or, the result is the Boolean XOR of the source register contents and the destination register contents, and is written back to the destination register Z - set if the result is zero N - set if the result is negative C - not defined

DSP Flags Register F1A100 Read/Write

This register provides status and control bit for several important DSP functions Control bits are

0	ZERO_FLAG	The ALU zero flag, set if the result of the last arithmetic operation was zero. Certain arithmetic instructions do not affect the flags, see above
1	CARRY_FLAG	The ALU carry flag, set or cleared by carry/borrow out of the adder/subtract, and reflects carry out of some shift operations, but it is not defined after other arithmetic operations
2	NEGA_FLAG	The ALU negative flag, set if the result of the last arithmetic operation was negative

3	IMASK	Interrupt mask, set by the interrupt control logic at the start of the service routine, and is cleared by the interrupt service routine writing a 0. Writing a 1 to this location has no effect
4-8	INT_ENA0-4	Interrupt enable bits for interrupts 0-4. The status of these bits is overridden by IMASK.
9-13	INT_CLR0-4	Interrupt latch clear bits for interrupts 0-4. These bits are used to clear the interrupt latches, which may be read from the status register. Writing a zero to any of these bits leaves it unchanged, and the read value is always zero.
14	REGPAGE	Switches from register bank 0 to register bank 1. This function is overridden by the IMASK flag, which forces register bank 0 to be used.
15	DMAEN	When DMAEN is set, DSP LOAD and STORE instructions perform external memory transfers at DMA priority, rather than GPU priority. This has no effect on program data fetches, which continue at GPU priority. This bit must **not** be changed while an external memory cycle is active. Note that these occur in the background, so be very careful about changing this flag dynamically, and do not modify it in an interrupt service routine.
16	INT_ENA5	Interrupt enable bit for interrupt 5. Function as bits 4-8.
17	INT_CLR5	Interrupt latch clear bit for interrupt 5. Function as bits 9-13.

WARNING - writing a value to the flag bits and making use of those flag bits in the following instruction will not work properly due to pipe-lining effects. If it is necessary to use flags set by a STORE instruction, then ensure that at least one other instruction lies between the STORE and the flags dependent instruction.

DSP Matrix Control Register F1A104 Write only

This register controls the function of the MMULT instruction. Control bits are:

0-3	MWIDTH	Matrix width, in the range 3 to 15
4	MADDW	When set, this control bit make the matrix held in memory be accessed down one column, as opposed to along one row.

DSP Matrix Address Register F1A108 Write only

This register determines where, in local RAM, the matrix held in memory is.

2-11	MTXADDR	Matrix address.

DSP Data Organisation Register F1A10C Write only

This register controls the physical layout of the DSP I/O registers and instructions. If its current contents are unknown, the same data should be written to both the low and high 16-bits.

0	BIG_IO	When this bit is set, 32-bit registers in the CPU I/O space are big-endian, i.e. the more significant 16-bits appear at the lower address.

2	BIG_INSTR	Normally, instructions are executed from a long-word in the order low word then high word When this bit is set the execution ordering is reversed, i e high word then low word However, move immediate data remains little-endian, i e the data must always be in the order low word then high word in the instruction stream

DSP Program Counter F1A110 Read/Write

The DSP program counter may be written whenever the DSP is idle (DSPGO is clear) This is normally used by the CPU to govern where program execution will start when the DSPGO bit is set

The DSP program counter may be read at any time, and will give the address of the instruction currently being executed If the DSP reads it, this must be performed by the MOVE PC,Rn instruction, and not by performing a load from it

The DSP program counter must always be written to before setting the DSPGO control bit. When the DSPGO bit is cleared, the program counter value will be corrupted, as at this point the pre-fetch queue is discarded.

DSP Control/Status Register F1A114 Read/Write

This register governs the interface between the CPU and the DSP

0	DSPGO	This bit stops and starts the DSP The CPU or DSP may write to this register at any time, but only the DSP should be used to clear this bit (unless single-stepping is enabled)
1	CPUINT	Writing a 1 to this bit allows the DSP to interrupt the CPU There is no need for any acknowledge, and no need to clear the bit to zero Writing a zero has no effect A value of zero is always read
2	DSPINT0	Writing a 1 to this bit causes a DSP interrupt type 0 There is no need for any acknowledge, and no need to clear the bit to zero Writing a zero has no effect A value of zero is always read
3	SINGLE_STEP	When this bit is set DSP single-stepping is enabled This means that program execution will pause after each instruction, until a SINGLE_GO command is issued The read status of this flag, SINGLE_STOP, indicates whether the DSP has actually stopped, and should be polled before issuing a further single step command A one means the DSP is awaiting a SINGLE_GO command
4	SINGLE_GO	Writing a one to this bit advances program execution by one instruction when execution is paused in single-step mode Neither writing to this bit at any other time, nor writing a zero, will have any effect Zero is always read
5	unused	Write zero
6-10	INT_LAT0-4	Interrupt latches for interrupts 0-4 The status of these bits indicate which interrupt request latch is currently active, and the appropriate bit should be cleared by the interrupt service routine, using the INT_CLR bits in the flags register Writing to these bits has no effect

11	BUS_HOG	When the DSP is executing code out of external RAM it will normally give up the bus between program fetches. This behaviour should allow the CPU to continue to run at the same time. Setting this bit causes the DSP to attempt to hold on to the bus between program fetches, which improves its execution speed, at the expense of any lower priority device using the bus.
12-15	VERSION	These bits allow the DSP version code to be read. Current version codes are: 2 First production release Future variants of the DSP may contain additional features or enhancements, and this value allows software to remain compatible with all versions. It is intended that future versions will be a superset of this DSP.
16	INT_LAT5	Interrupt latch for interrupt 5. Has the same function for interrupt 5 as bits 6-10 have for interrupts 0-4.

Modulo instruction mask — F1A118 — Write only

This 32-bit register holds the value which governs which bits are modified by the ADDQMOD and SUBQMOD instructions. A 1 means that the bit will be unaffected, a 0 means that it may be changed. Normally, the higher bits are set to 1 and the lower bits to 0. This allows addresses to be readily generated for circular buffers of size 2^n bytes, where n is between 0 and 31.

Divide unit remainder — F1A11C — Read only

This 32-bit register contains a value from which the remainder after a division may be calculated. Refer to the section on the Divide Unit.

Divide unit Control — F1A11C — Write only

1	DIV_OFFSET	If this bit is set, then the divide unit performs division of unsigned 16.16 bit numbers, otherwise 32-bit unsigned integer division is performed.

Multiply & Accumulate High Result Bits — F1A120 — Read only

This 32-bit register allows the high eight bits of the accumulated result to be read. After a RESMAC instruction the result register of the RESMAC contains the bottom 32 bits of the accumulated value, and this register contains the top eight bits, which are sign-extended to 32 bits.

In the DSP, certain peripheral IO functions are mapped into the internal DSP space for higher efficiency when the DSP is controlling them. These are effectively 32-bit locations. These are the PWM DACs and the Synchronous Serial Interface.

Writing Fast DSP Programs

Refer to the section entitled 'Writing Fast GPU Programs'. The same rules apply to the DSP.

Tom and Jerry Hardware Interface

This section discusses the hardware interface to the Tom and Jerry devices

Pinout

TOM Pinout

1	VSS1	0V to output pads	Supply pin
2	VDD1	5V to output pads	Supply pin
3	XR0	2mA output	Video output
4	XR1	2mA output	Video output
5	XR2	2mA output	Video output
6	XR3	2mA output	Video output
7	XR4	2mA output	Video output
8	XR5	2mA output	Video output
9	XR6	2mA output	Video output
10	VDD1	5V to output pads	Supply pin
11	XR7	2mA output	Video output
12	XG0	2mA output	Video output
13	XG1	2mA output	Video output
14	XG2	2mA output	Video output
15	VSS2	0V to internal logic	Supply pin
16	XG3	2mA output	Video output
17	XG4	2mA output	Video output
18	XG5	2mA output	Video output
19	XG6	2mA output	Video output
20	XG7	2mA output	Video output
21	XB0	2mA output	Video output
22	XB1	2mA output	Video output
23	XB2	2mA output	Video output
24	XB3	2mA output	Video output
25	XB4	2mA output	Video output
26	XB5	2mA output	Video output
27	XB6	2mA output	Video output
28	XB7	2mA output	Video output
29	VSS1	0V to output pads	Supply pin
30	VDD1	5V to output pads	Supply pin
31	XHSL	2mA output/TTL input	Video horizontal synchronization
32	XVSL	2mA output/TTL input	Video vertical synchronization
33	XLP	CMOS input	Light-pen input
34	XINC	2mA output	Video encrustation control
35	XEXPL	4mA output	Expansion bus enable
36	XFC0	2mA output/TTL input	CPU function code
37	XFC1	2mA output/TTL input	CPU function code

38	VSS1	0V to output pads	Supply pin
39	XFC2	2mA output/TTL input	CPU function code
40	XDREQL	2mA output/TTL input	CPU transfer request
41	XDTACKL	2mA output	CPU transfer acknowledge
42	XRW	2mA output/TTL input	Bus transfer direction
43	VDD1	5V to output pads	Supply pin
44	XSIZ0	2mA output/TTL input	Bus transfer size
45	XSIZ1	2mA output/TTL input	Bus transfer size
46	XINTL	2mA output	CPU interrupt output
47	XDINT	CMOS input	DSP interrupt input
48	VSS3	0V to input pads	Supply pin
49	XBRL	2mA output/TTL input	CPU bus request
50	XBGL	CMOS input	CPU bus grant
51	XBGA	2mA output/TTL input	CPU bus grant acknowledge
52	XDSPCSL	2mA output	DSP chip select
53	XRESETL	CMOS input	Master reset
54	VDD3	5V to input pads	Supply pin
55	XTEST	CMOS input	Test pin
56	XWAITL	CMOS input	Expansion bus wait request
57	XROMCSL1	2mA output	ROM chip select for cartridge
58	XROMCSL0	2mA output	ROM chip select for boot-strap
59	XDBGL	2mA output	DSP bus grant
60	VSS3	0V to input pads	Supply pin
61	VDD3	5V to input pads	Supply pin
62	XDBRL1	CMOS input	DSP bus request priority level 0
63	XDBRL0	CMOS input	DSP bus request priority level 1
64	XPCLK	CMOS input	Internal processor clock
65	VSS2	0V to internal logic	Supply pin
66	XVCLK	CMOS input	Video clock
67	XMASKA0	2mA output	Address line for memory
68	XMASKA1	2mA output	Address line for memory
69	XMASKA2	2mA output	Address line for memory
70	XA0	4mA output/TTL input	System address bus
71	XA1	4mA output/TTL input	System address bus
72	XA2	4mA output/TTL input	System address bus
73	XA3	4mA output/TTL input	System address bus
74	XA4	4mA output/TTL input	System address bus
75	XA5	4mA output/TTL input	System address bus
76	XA6	4mA output/TTL input	System address bus
77	VSS3	0V to input pads	Supply pin
78	XA7	4mA output/TTL input	System address bus
79	XA8	4mA output/TTL input	System address bus
80	XA9	4mA output/TTL input	System address bus
81	XA10	4mA output/TTL input	System address bus
82	XA11	4mA output/TTL input	System address bus
83	XA12	4mA output/TTL input	System address bus
84	XA13	4mA output/TTL input	System address bus

85	XA14	4mA output/TTL input	System address bus
86	XA15	4mA output/TTL input	System address bus
87	XA16	4mA output/TTL input	System address bus
88	XA17	4mA output/TTL input	System address bus
89	XA18	4mA output/TTL input	System address bus
90	XA19	4mA output/TTL input	System address bus
91	XA20	4mA output/TTL input	System address bus
92	XA21	4mA output/TTL input	System address bus
93	VSS3	0V to input pads	Supply pin
94	XA22	4mA output/TTL input	System address bus
95	XA23	4mA output/TTL input	System address bus
96	VSS1	0V to output pads	Supply pin
97	VDD1	5V to output pads	Supply pin
98	XD24	4mA output/TTL input	System data bus
99	XD23	4mA output/TTL input	System data bus
100	XD8	8mA output/TTL input	System data bus
101	XD7	8mA output/TTL input	System data bus
102	XD25	4mA output/TTL input	System data bus
103	XD22	4mA output/TTL input	System data bus
104	VDD3	5V to input pads	Supply pin
105	XD9	8mA output/TTL input	System data bus
106	XD6	8mA output/TTL input	System data bus
107	XD26	4mA output/TTL input	System data bus
108	XD21	4mA output/TTL input	System data bus
109	XD10	8mA output/TTL input	System data bus
110	XD5	8mA output/TTL input	System data bus
111	XD27	4mA output/TTL input	System data bus
112	VSS3	0V to input pads	Supply pin
113	XD20	4mA output/TTL input	System data bus
114	VDD1	5V to output pads	Supply pin
115	XD11	8mA output/TTL input	System data bus
116	XD4	8mA output/TTL input	System data bus
117	XD28	4mA output/TTL input	System data bus
118	XD19	4mA output/TTL input	System data bus
119	VSS2	0V to internal logic	Supply pin
120	XD12	8mA output/TTL input	System data bus
121	XD3	8mA output/TTL input	System data bus
122	XD29	4mA output/TTL input	System data bus
123	XD18	4mA output/TTL input	System data bus
124	XD13	8mA output/TTL input	System data bus
125	XD2	8mA output/TTL input	System data bus
126	XD30	4mA output/TTL input	System data bus
127	XD17	4mA output/TTL input	System data bus
128	XD14	8mA output/TTL input	System data bus
129	VSS1	0V to output pads	Supply pin
130	XD1	8mA output/TTL input	System data bus
131	XD31	4mA output/TTL input	System data bus

132	VSS3	0V to input pads	Supply pin
133	VDD3	5V to input pads	Supply pin
134	XD16	4mA output/TTL input	System data bus
135	XD15	8mA output/TTL input	System data bus
136	XD0	8mA output/TTL input	System data bus
137	XMA10	16mA output/TTL input	DRAM multiplexed address bus
138	XMA9	16mA output/TTL input	DRAM multiplexed address bus
139	XMA8	16mA output/TTL input	DRAM multiplexed address bus
140	VSS1	0V to output pads	Supply pin
141	XMA7	16mA output/TTL input	DRAM multiplexed address bus
142	VSS1	0V to output pads	Supply pin
143	XMA6	16mA output/TTL input	DRAM multiplexed address bus
144	XMA5	16mA output/TTL input	DRAM multiplexed address bus
145	XMA4	16mA output/TTL input	DRAM multiplexed address bus
146	XMA3	16mA output/TTL input	DRAM multiplexed address bus
147	VDD1	5V to output pads	Supply pin
148	XMA2	16mA output/TTL input	DRAM multiplexed address bus
149	XMA1	16mA output/TTL input	DRAM multiplexed address bus
150	XMA0	16mA output/TTL input	DRAM multiplexed address bus
151	XD40	4mA output/TTL input	System data bus
152	VSS3	0V to input pads	Supply pin
153	XD39	4mA output/TTL input	System data bus
154	XD56	4mA output/TTL input	System data bus
155	XD55	4mA output/TTL input	System data bus
156	XD41	4mA output/TTL input	System data bus
157	XD38	4mA output/TTL input	System data bus
158	XD57	4mA output/TTL input	System data bus
159	XD54	4mA output/TTL input	System data bus
160	XD42	4mA output/TTL input	System data bus
161	VDD3	5V to input pads	Supply pin
162	XD37	4mA output/TTL input	System data bus
163	XD58	4mA output/TTL input	System data bus
164	VSS3	0V to input pads	Supply pin
165	VDD3	5V to input pads	Supply pin
166	XD53	4mA output/TTL input	System data bus
167	VSS1	0V to output pads	Supply pin
168	XD43	4mA output/TTL input	System data bus
169	XD36	4mA output/TTL input	System data bus
170	VSS2	0V to internal logic	Supply pin
171	XD59	4mA output/TTL input	System data bus
172	XD52	4mA output/TTL input	System data bus
173	XD44	4mA output/TTL input	System data bus
174	XD35	4mA output/TTL input	System data bus
175	XD60	4mA output/TTL input	System data bus
176	XD51	4mA output/TTL input	System data bus
177	XD45	4mA output/TTL input	System data bus
178	XD34	4mA output/TTL input	System data bus

179	XD61	4mA output/TTL input	System data bus
180	XD50	4mA output/TTL input	System data bus
181	XD46	4mA output/TTL input	System data bus
182	XD33	4mA output/TTL input	System data bus
183	XD62	4mA output/TTL input	System data bus
184	XD49	4mA output/TTL input	System data bus
185	XD47	4mA output/TTL input	System data bus
186	XD32	4mA output/TTL input	System data bus
187	XD63	4mA output/TTL input	System data bus
188	VSS1	0V to output pads	Supply pin
189	XD48	4mA output/TTL input	System data bus
190	XWEL0	16mA output	Memory write strobe
191	XWEL1	16mA output	Memory write strobe
192	XWEL2	4mA output	Memory write strobe
193	XWEL3	4mA output	Memory write strobe
194	XWEL4	4mA output	Memory write strobe
195	XWEL5	4mA output	Memory write strobe
196	XWEL6	4mA output	Memory write strobe
197	XWEL7	4mA output	Memory write strobe
198	XOEL0	16mA output	Memory output enable
199	XOEL1	8mA output	Memory output enable
200	VSS1	0V to output pads	Supply pin
201	VDD1	5V to output pads	Supply pin
202	XOEL2	8mA output	Memory output enable
203	XRASL0	16mA output	DRAM bank 0 row address strobe
204	XRASL1	16mA output	DRAM bank 1 row address strobe
205	XCASL0	16mA output	DRAM bank 0 column address strobe
206	XCASL1	16mA output	DRAM bank 1 column address strobe
207	VSS3	0V to input pads	Supply pin
208	VSS1	0V to output pads	Supply pin

JERRY Pinout

1	XVCLK	8mA fast output/TTL input	Video clock output
2	XPCLKOSC	CMOS input	Processor clock input from oscillator
3	XPCLKOUT	8mA fast output	Processor clock output to system
4	XPCLKIN	CMOS input	Processor clock input to logic
5	VSS1	0V to output pads	Supply pin
6	XVCLKDIV	8mA output	Video clock divide output for a PLL
7	XDSPCSL	CMOS input	DSP chip select
8	XPCLKDIV	8mA output	Processor clock divide output for a PLL
9	VSS1	0V to output pads	Supply pin
10	XCHRIN	OSC4CI	Chroma crystal oscillator input
11	XCHROUT	OSC4CO	Chroma crystal oscillator output
12	VDD1	5V to output pads	Supply pin
13	XCHRDIV	8mA output	Chroma oscillator divide output for a PLL

14	VSS3	0V to input pads	Supply pin
15	XDTACKL	CMOS input	Bus master transfer acknowledge
16	XRW	8mA tri-state output	Bus master transfer direction
17	XSIZ_0	8mA tri-state output	Bus master transfer size
18	VDD3	5V to input pads	Supply pin
19	VSS3	0V to input pads	Supply pin
20	XSIZ_1	8mA tri-state output	Bus master transfer size
21	XCPUCLK	8mA fast output	CPU clock
22	XOEL0	CMOS input	Bus slave read enable
23	XWEL0	CMOS input	Bus slave write enable
24	XDINT	8mA output	DSP interrupt
25	XDBRL_0	8mA output	DSP bus request priority level 0
26	XDBRL_1	8mA output	DSP bus request priority level 1
27	XDBGL	CMOS input	DSP bus grant
28	XRESETIL	CMOS input,	Reset input from reset circuit
29	XRESETL	8mA output	Reset output for rest of system
30	XTEST	CMOS input	Test pin
31	XDREQL	8mA tri-state output	Bus master transfer request
32	XIORDL	8mA output	Expansion bus IO read strobe
33	VSS1	0V to output pads	Supply pin
34	VDD1	5V to output pads	Supply pin
35	XIOWRL	8mA output	Expansion bus IO write strobe
36	XEINT_0	CMOS input	Expansion bus interrupt 0
37	XEINT_1	CMOS input	Expansion bus interrupt 1
38	VSS3	0V to input pads	Supply pin
39	VDD3	5V to input pads	Supply pin
40	XGPIOL_0	8mA output/TTL input	General purpose expansion IO address decode
41	XGPIOL_1	8mA output/TTL input	General purpose expansion IO address decode
42	XGPIOL_2	8mA output/TTL input	General purpose expansion IO address decode
43	XGPIOL_3	8mA output/TTL input	General purpose expansion IO address decode
44	VSS2	0V to internal logic	Supply pin
45	XGPIOL_4	8mA output	General purpose expansion IO address decode
46	XGPIOL_5	8mA output	General purpose expansion IO address decode
47	VSS1	0V to output pads	Supply pin
48	XJOY_0	8mA output/TTL input	Joystick interface control
49	XJOY_1	8mA output/TTL input	Joystick interface control
50	XJOY_2	8mA output/TTL input	Joystick interface control
51	XJOY_3	8mA output/TTL input	Joystick interface control
52	XSERIN	CMOS input	Asynchronous serial input
53	VDD1	5V to output pads	Supply pin
54	VDD1	5V to output pads	Supply pin
55	VSS1	0V to output pads	Supply pin
56	XSEROUT	8mA output	Asynchronous serial output
57	VSS3	0V to input pads	Supply pin
58	XSCK	8mA output/TTL input	Synchronous serial clock
59	XWS	8mA output/TTL input	Synchronous serial word select
60	XI2STXD	8mA output	Synchronous serial data out

61	XI2SRXD	CMOS input	Synchronous serial data in
62	VSS1	0V to output pads	Supply pin
63	VDD1	5V to output pads	Supply pin
64	VSS1	0V to output pads	Supply pin
65	XLDAC_0	8mA output	PWM DAC output
66	XLDAC_1	8mA output	PWM DAC output
67	XRDAC_0	8mA output	PWM DAC output
68	XRDAC_1	8mA output	PWM DAC output
69	VSS1	0V to output pads	Supply pin
70	VDD1	5V to output pads	Supply pin
71	XD_31	8mA output/TTL input	System data bus
72	VDD3	5V to input pads	Supply pin
73	XD_30	8mA output/TTL input	System data bus
74	XD_29	8mA output/TTL input	System data bus
75	XD_28	8mA output/TTL input	System data bus
76	XD_27	8mA output/TTL input	System data bus
77	XD_26	8mA output/TTL input	System data bus
78	VSS3	0V to input pads	Supply pin
79	XD_25	8mA output/TTL input	System data bus
80	XD_24	8mA output/TTL input	System data bus
81	XD_23	8mA output/TTL input	System data bus
82	XD_22	8mA output/TTL input	System data bus
83	XD_21	8mA output/TTL input	System data bus
84	XD_20	8mA output/TTL input	System data bus
85	XD_19	8mA output/TTL input	System data bus
86	XD_18	8mA output/TTL input	System data bus
87	XD_17	8mA output/TTL input	System data bus
88	XD_16	8mA output/TTL input	System data bus
89	XD_15	8mA output/TTL input	System data bus
90	VDD1	5V to output pads	Supply pin
91	VSS2	0V to internal logic	Supply pin
92	VSS3	0V to input pads	Supply pin
93	XD_14	8mA output/TTL input	System data bus
94	XD_13	8mA output/TTL input	System data bus
95	XD_12	8mA output/TTL input	System data bus
96	XD_11	8mA output/TTL input	System data bus
97	XD_10	8mA output/TTL input	System data bus
98	XD_9	8mA output/TTL input	System data bus
99	XD_8	8mA output/TTL input	System data bus
100	XD_7	8mA output/TTL input	System data bus
101	XD_6	8mA output/TTL input	System data bus
102	XD_5	8mA output/TTL input	System data bus
103	VSS1	0V to output pads	Supply pin
104	XD_4	8mA output/TTL input	System data bus
105	VSS3	0V to input pads	Supply pin
106	XD_3	8mA output/TTL input	System data bus
107	XD_2	8mA output/TTL input	System data bus

108	XD_1	8mA output/TTL input	System data bus
109	XD_0	8mA output/TTL input	System data bus
110	XA_23	8mA output/TTL input	System address bus
111	XA_22	8mA output/TTL input	System address bus
112	VDD3	5V to input pads	Supply pin
113	XA_21	8mA output/TTL input	System address bus
114	VSS1	0V to output pads	Supply pin
115	XA_20	8mA output/TTL input	System address bus
116	VSS3	0V to input pads	Supply pin
117	XA_19	8mA output/TTL input	System address bus
118	XA_18	8mA output/TTL input	System address bus
119	XA_17	8mA output/TTL input	System address bus
120	VDD3	5V to input pads	Supply pin
121	XA_16	8mA output/TTL input	System address bus
122	XA_15	8mA output/TTL input	System address bus
123	XA_14	8mA output/TTL input	System address bus
124	XA_13	8mA output/TTL input	System address bus
125	VSS1	0V to output pads	Supply pin
126	VDD1	5V to output pads	Supply pin
127	VSS1	0V to output pads	Supply pin
128	XA_12	8mA output/TTL input	System address bus
129	XA_11	8mA output/TTL input	System address bus
130	XA_10	8mA output/TTL input	System address bus
131	XA_9	8mA output/TTL input	System address bus
132	XA_8	8mA output/TTL input	System address bus
133	XA_7	8mA output/TTL input	System address bus
134	XA_6	8mA output/TTL input	System address bus
135	VSS3	0V to input pads	Supply pin
136	VSS1	0V to output pads	Supply pin
137	VDD3	5V to input pads	Supply pin
138	XA_5	8mA output/TTL input	System address bus
139	XA_4	8mA output/TTL input	System address bus
140	XA_3	8mA output/TTL input	System address bus
141	XA_2	8mA output/TTL input	System address bus
142	XA_1	8mA output/TTL input	System address bus
143	XA_0	8mA output/TTL input	System address bus
144	VSS3	0V to input pads	Supply pin

TOM Pin Description

<table>
<tr><td>XD[0 63]</td><td>The main data bus Connects to DRAM, Jerry and 68000 Isolated from slower logic with TTL TOM may simultaneously drive parts of the bus while inputting on others This allows 16 and 32 bit processors to work with 64 bit DRAM Narrower peripherals should be placed on the less significant end of the data bus XD[0 15] are 8mA, XD[16 63] are 4mA outputs</td></tr>
<tr><td>XA[0 23]</td><td>The main address bus Connects to Jerry and the 68000 Isolated from slower logic with TTL Narrow memory devices (less than 64bit) should not be connected to XA[0 2] but to XMASKA[0 2] This allows TOM to break one wide request into several narrower cycles at different addresses These are 4mA outputs</td></tr>
<tr><td>XMA[0 10]</td><td>Multiplexed address bus These signals carry the address
to the DRAMs The actual address signals to which each
relates depends on the width of DRAM, the number of
columns in the DRAM and whether outputting the row
address or the column address These are 16mA outputs

During reset these signals become inputs and 1K resistors tied either to ground or +5V are used to configure aspects of the system which cannot be set by software They should be tied as follows
<table>
<tr><td>XMA[0]</td><td>romhi</td><td>+5V</td></tr>
<tr><td>XMA[1]</td><td>romwidth[0]</td><td>0V</td></tr>
<tr><td>XMA[2]</td><td>romwidth[1]</td><td>0V</td></tr>
<tr><td>XMA[4]</td><td>nocpu</td><td>+5V</td></tr>
<tr><td>XMA[5]</td><td>cpu32</td><td>0V</td></tr>
<tr><td>XMA[6]</td><td>bigend</td><td>+5V</td></tr>
<tr><td>XMA[7]</td><td>extclk</td><td>0V</td></tr>
<tr><td>XMA[8]</td><td>68k</td><td>+5V</td></tr>
</table></td></tr>
<tr><td>XMASKA[0 2]</td><td>Least significant address output These are incremented when TOM breaks a wide cycle request into several narrow cycles at different addresses These are 2mA outputs</td></tr>
<tr><td>XROMCSL[0 1]</td><td>ROM chip selects Active low 2mA outputs</td></tr>
<tr><td>XRASL[0 1]</td><td>Row address strobes for each of two banks of DRAM Once asserted (active low) each RAS remains asserted until an access from another row or a refresh cycle These are 16mA outputs</td></tr>
<tr><td>XCASL[0 1]</td><td>Column address strobes for each of two banks of DRAM These are 16mA outputs.</td></tr>
<tr><td>XOEL[0 2]</td><td>Memory output enables XOEL[0] applies to XD[0 15] and is a 16mA output, XOEL[1] applies to XD[16 32] and is an 8mA output, XOEL[2] applies to XD[32 63] and is an 8mA output XOEL[0 1] should be used to control the direction of the data bus transceivers.</td></tr>
<tr><td>XWEL[0 7]</td><td>Memory write enables XWEL[0] applies to XD[0 7], XWEL[1] applies to XD[8 15] and so on These are 16mA outputs</td></tr>
</table>

XPCLK	Processor clock input. This is the main clock used by the memory interface, object processor, graphics processor and blitter. The clock high time defines the CAS precharge time so the mark space should be controlled (most crystal oscillators are OK).
XVCLK	Video clock input This clock is used by the video time-base and pixel logic. It should be identical to or somewhat slower than the processor clock XPCLK. The video subsystem invokes the object processor by generating a pulse one video clock cycle wide. This is sampled by the processor clock. In order to guarantee that the pulse is seen the clocks should be identical or the video clock period should be greater by at least a few nanoseconds in order to satisfy sample and hold requirements and avoid problems relating to pulse thinning and clock jitter.
XRESETL	Active low reset input. Not a Schmitt input
XWAITL	Active low wait input. Can be used to add wait states to memory and peripheral transfers This input is tested on the rising clock edge prior to the last cycle in a transfer. DRAM transfers may not have wait states.
XDREQL	Active low transfer request. Used by external bus masters (68000 and Jerry) to request a memory cycle. This signal is connected to the 68000's address strobe. When internal bus masters own the bus This signal is asserted during the first cycle of all transfers. This is a 2mA output.
XDTACKL	Active low transfer acknowledge. Used to signal to external bus masters that the cycle has completed. This signal is maintained until XDREQL is retracted. Read data is presented by TOM at the same time as XDTACKL. This is a 2mA output.
XRW	Read/write. This determines the direction of the current transfer. Driven by internal bus masters when they own the bus This is a 2mA output
XSIZ[0 1]	Transfer size. These determine the number of bytes to be transferred. They are connected to the 68000's LDS and UDS outputs so they also imply a[0] when the 68000 owns the bus. They are 2mA outputs. When Jerry or another external non 68000 microprocessor owns the bus they mean the following:

XSIZ[1]	XSIZ[0]	bytes
0	0	4
0	1	1
1	0	2
1	1	3

When an internal bus master owns the bus they become outputs and mean the following:

XSIZ[1]	XSIZ[0]	bytes
0	0	8
0	1	1
[illegible]	0	2
1	1	4

XDBRL[0..1]	Jerry bus request inputs. These two inputs request the bus for Jerry at one of two priorities. XDBRL[0] requests the bus at a priority just less than video. XDBRL[1] requests the bus at a priority greater than video but less than refresh.
XDBGL	Jerry bus grant output. Active low 2mA output.
XEXPL	Active low expansion bus enable. This 4mA output enables the data bus transceivers for all transfers to ROMs and peripherals. By dividing the data bus into a fast part and a slow part the parasitic capacitance can be reduced to keep speed high. This scheme also reduces the likelihood of static damage to the ASICs or DRAM.
XDSPCSL	Active low Jerry chip select. This 2mA output is asserted by TOM for all transfers in Jerry's 64k address range.
XINTL	Active low interrupt 2mA output. Used to interrupt the 68000.
XHSL, XVSL	Active low horizontal and vertical video syncs. May be programmed to output composite sync on XVSL. These are 2mA outputs. These may also be used as inputs so that external active low syncs can reset the internal vertical and horizontal time-bases in order to facilitate rapid genlocking.
XLP	Light pen input.
XR[0..7] XG[0..7] XB[0..7]	Red, green and blue outputs. These should be connected eight bit DACs to generate the analogue RGB required by monitors and video encoders. In practice an R-2R ladder can be directly attached to these outputs. These are 2mA outputs.
XINCL	Incrust output. This 2mA output may be used to switch between the internally generated video and an external video source on a pixel by pixel basis. The switch must be provided externally.
XDINT	Jerry interrupt input. Interrupts from Jerry are funnelled through this to the 68000.
XFC[0..2]	68000 function code signals. If the microprocessor is a 68000 then these inputs are used to qualify transfer requests and decode interrupt acknowledge cycles. When an internal bus master owns the bus the value 101 is output on these 2mA outputs.
XBRL	68000 bus request. This 2mA output is used to request the bus from the 68000. May also be used as an input for external bus masters.
XBGL	68000 bus grant input.
XBA	68000 bus grant acknowledge 2mA output.
XTEST	Test input. This is used for testing the chip in production.

Jerry Pin Description

XDSPCSL	DSP chip select input Active low signal indicates when Jerry is being addressed Jerry occupies 64k memory locations The chip select input allows multiple Jerry systems
XPCLKOSC	Processor clock oscillator input This input does not clock Jerry but clocks two dividers The first programmable divider divides by between 1 and 1024 The output, pclkdiv, may be used in a phase locked loop to synthesize the processor clock from a convenient reference frequency The second divider is an optional divide by two If xjoy[2] is pulled high during reset then pclkout is half the frequency of pclkosc This may be used to give pclkout a well defined duty cycle The divider does not drive Jerry's clock directly but must first go off-chip and re-enter via the pclkin pin This minimises clock skew between Tom & Jerry and allows an external fix to any clock skew problem
XPCLKIN	This is the main clock input to Jerry
XDBGL	Active low DSP bus grant input When asserted the DSP must drive the 68000 bus control signals and may perform transfers to-from memory
XOEL[0]	Active low output enable input Enables Jerry data when being read Also used in the generation of joystick read strobes
XWEL[0]	Active low write enable input Latches write data into Jerry when being written Also used in the generation of joystick write strobes
XSERIN	Uart data input Programmable polarity
XDTACKL	Active low data transfer acknowledge input Output by Tom to mark the end of the current transfer
XI2SRXD	I^2S serial data input
XEINT[0 1]	External interrupt inputs A rising edge on eint[0] may generate an interrupt to the 68000 or the DSP A rising edge on eint[1] may interrupt the DSP and is intended to implement a DMA mechanism
XTEST	Test input for chip testing

XCHRIN	Chroma oscillator input This input and the corresponding output xchrout may be used as a crystal oscillator The oscillator may typically be used in one of two ways A crystal with a frequency equal to the colour subcarrier is used. This is divided by a programmeable divider to the xchrdiv output This frequency is used as a reference by an external phase locked loop in the generation of the video clock This provides a flexible video clock which is tied to the colour subcarrier The colour subcarrier may be taken from the xchrout output. A crystal with a frequency which is an integer multiple of the colour subcarrier frequency is used This is the video clock frequency The programmable divider is programmed to the multiple and the colour subcarrier is output on xchrdiv This provides a cheaper but less flexible video clock which is tied to the colour subcarrier
XRESETIL	Active low reset input
XD[0 31]	Jerry's bidirectional data bus Attached to Tom, 68000 and DRAM Because Tom treats Jerry the same way as the microprocessor Jerry may only use the lower 16 bits of the data bus if the microprocessor is 16 bits If xjoy[0] is pulled high during reset then Jerry uses a 16 bit interface If pulled low Jerry uses a 32 bit interface 8mA outputs are used throughout
XA[0 23]	Jerry's bidirectional address bus Driven by Jerry when xdbgl is asserted 8mA outputs
XJOY[0 3]	Joystick control outputs These 8mA outputs are used as follows
XJOY[0]	Active low output enables the 16 joystick inputs onto the data bus Pulled high during reset to force Jerry to use a 16 bit interface Pulled low for a 32 bit interface
XJOY[1]	Active low output enables the four button inputs onto the data bus Pulled high during reset for big endian (Motorola) operation, low for little endian (Intel) operation
XJOY[2]	Active low output latches data from the bottom eight bits of the data bus into the joystick output latch Pulled high during reset to divide the pclkosc input by two in order to get a 50% duty cycle on the main clock Pulled low there is no divide
XJOY[3]	Active low output enables the outputs of the joystick output latch Pulled high during reset to disable internal clock shaping logic in case of a design fault Pull low for normal operation

<table>
<tr><td>XGPIOL[0 5]</td><td>General purpose IO decode outputs These active low 8mA
outputs are asserted for certain ranges of IO addresses
Intended to reduce the amount of logic required to
interface external peripherals
XGPIOL[0 2] are used as inputs during reset but have no purpose on this version of the ASIC</td></tr>
<tr><td>XSCK, XWS</td><td>I²S clock and word select These may be programmed as inputs or as outputs Depending on whether Jerry is I²S slave or master 8mA outputs</td></tr>
<tr><td>XVCLK</td><td>Video clock input or output This pin may be programmed
as an 8mA output in which case it simply buffers the
crystal oscilator It may be programmed as an input in
which case the input is divided by a programmable
divider The output xvclkdiv may be used in a phase
locked loop to synthesize the video clock from a
fraction of the colour subcarrier
Programmed as an input on reset</td></tr>
<tr><td>XSIZ[0 1]</td><td>Transfer size These determine the number of bytes to be transfered They are connected to the 68000's lds and uds outputs These 8mA outputs are enabled when xdbgl is asserted They mean the following -
<table>
<tr><td>siz[1]</td><td>siz[0]</td><td>bytes</td></tr>
<tr><td>0</td><td>0</td><td>4</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>2</td></tr>
<tr><td>1</td><td>1</td><td>3</td></tr>
</table></td></tr>
<tr><td>XRW</td><td>Transfer direction 8mA output driven when xdbgl asserted High for reads</td></tr>
<tr><td>XDREQL</td><td>Transfer request 8mA active low output driven when xdbgl is asserted Connects to Tom and 68000 address strobe</td></tr>
<tr><td>XDBRL[0 1]</td><td>Dsp bus requests Active low 8mA outputs xdbrl[0] requests the bus at a priority just less than video xdbrl[1] requests the bus at a priority greater than video but less than refresh</td></tr>
<tr><td>XINT</td><td>Active high 8mA interrupt output All Jerry interrupts will assert this signal which connects to Tom</td></tr>
<tr><td>XSEROUT</td><td>Uart data output, programmable polarity 8mA drive</td></tr>
<tr><td>XVCLKDIV</td><td>Video clock divider output 8mA drive, see xvclk</td></tr>
<tr><td>XCHRDIV</td><td>Colour subcarrier divider output, 8mA drive see xchrin</td></tr>
</table>

XPCLKOUT	Main system clock output. Fast 8mA drive. Buffers and optionally divides xplckosc by two.
XPCLKDIV	Processor clock divider output, 8mA drive, see xpclkosc.
XRESETL	Active low reset output. 8mA output buffers xresetil.
XCHROUT	Crystal oscillator output, partner to xchrin.
XRDAC[0..1] XLDAC[0..1]	PWM outputs xrdac[0..1] are the right channel xldac[0..1] are the left channel. xrdac[0] and xldac[0] are the less significant outputs and are fed through resistors 128 times greater than those attached to xrdac[1] and xldac1[1] for summing. 8mA outputs
XIOWRL	IO write strobe. 8mA output is the OR of xdspcsl and xwel[0]. May be used to make peripheral attachment easier.
XIORDL	IO read strobe. 8mA output is the OR of xdspcsl and xoel[0]. may be used to make peripheral attachment easier.
XI2STXD	I^{2}S transmit data. 8mA output.
XCPUCLK	68000 clock output. Fast 8mA output. This outputs pclkout divided by two.

Timing Diagrams

ROM1 Timing

The following diagram shows a five cycle ROM1 read cycle without WAIT.

```
                   __    __    __    __    __    __    __    __    __    __
XPCLK           __/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \_
                ____                                              _________
ASL                 \__\_____________________________________________/__/
                ____________________________________________________________  _
XDTACKL                                                             \________/
                ___________________                                  __________
XROMCSL[1]                         \________________________________/
                ___________________                                  __________
XOEL[0..1]                         \________________________________/
                ___________________                                  __________
XEXPL                              \________________________________/
                                                              ______
DIN             ---------------------- <<<<<<<<<<<<<<<<<<<<<<<<______>-------------
                                                                      __________
DOUT            -----------------------------------------------------<__________>-
State              |  A  |  B  |  1  |  2  |  3  |  4  |  5  |  C  |  D  |
```

For a write cycle the write strobes have this timing

```
                ______________________                          _______________
XWEL[0..3]                            \________________________/
```

Explanation

Tom's memory controller is active during states 1 to 5 and idle during states A to D which have been labelled to clarify this discussion.

A) The 68000 presents an address, UDS, LDS and RW then drives AS low. AS is synchronised by TOM so as not to disrupt the memory controller state machine.

B) TOM decodes the address and determines the type of cycle required. Internal bus masters can pipeline requests so this phase can happen while a transfer is occurring. In that case there are no idle states and ROMCSL[I] remains asserted for successive accesses to that range of addresses.

I) TOM asserts XROMCSL[I], XEXPL and XOEL[0] or XOEL[I] (or both depending on the address and width of the transfer). The address

presented by the 68000 is buffered by TTL and applied to the ROM. If an internal bus master caused the cycle then TOM would drive the address bus and the address would change at the same time as these signals being asserted. The data bus buffers are enabled and turned in the data-in direction.

2-4) The memory controller waits a number of clock cycles determined by bits 3 & 4 of register MEMCON1. During this time the ROM data has time to settle, get through the TTL and settle on the main data-bus. The signal XWAITL is sampled by the rising clock edge at the end of state 4. If this is inactive the controller enters the final state. If active the controller enters a wait state. The controller samples XWAITL again at the end of the wait state and repeats until it is inactive when it enters the final state.

5) The data on the main data bus is routed through to the appropriate part of Tom's internal 64-bit data bus. The data is latched by the rising clock edge at the end of state 5. This same clock edge causes XROMCSL[1], XEXPL and XOEL[0..1].

C&D) When the 68000 is bus master XDTACKL is asserted by TOM and the internally latched data is enabled onto the external data bus. This allows the 68000 to read data from memory wider than 16 bits and also allows it to do 16 bit reads from 8 bit memory. AS is sampled again and the situation remains the same until the clock edge after AS is retracted.

The next diagram shows how XWAITL affects the transfer

```
               __    __    __    __    __    __    __    __    __    __
XPCLK       __/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \_
            __________                                          ________
XROMCSL[1]            \________________________________________/
            __________                                          ________
XOEL[0..1]            \________________________________________/
            __________                                          ________
XEXPL                 \________________________________________/
                                                      ___
XWAITL      ------------------------------___---___---   ----------------
                                                         _________
DIN         ---------------<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<_________>----
                                                                 _______
DOUT        ----------------------------------------------------<_______
```

```
State          |  B  |  1  |  2  |  3  |  4  |  W  |  W  |  5  |  C  |
```

For a write cycle the write strobes have this timing

```
                 _____________                                      __________
XWEL[0..3]                    \____________________________________/
```

XWAITL is sampled by the clock edge at the end of state 4 and at the end of wait states

If the ROM speed is longer than five clock cycles the above sequence still applies but there are correspondingly more cycles in the transfer XWAITL is always tested before the last cycle in the transfer

All outputs are synchronously generated and the delay from the corresponding XPCLK clock edge depends on the load capacitance and silicon processing Output drive strengths have been minimised and matched to the anticipated load in order to satisfy ASIC power pin rules With a 30pF load and worst case processing the delays from XPCLK input to various outputs are as follows

	Low to High	High to Low
ROMCSL[I]	32	50
XOEL[I]	28	3I
XDTACKL	30	43
XWEL[1]	22	2I

Most outputs follow the rising edge of XPCLK apart from XWEL[0 7] and the falling edge of XCASL[0 I] which follow the falling edge of XPCLK The inputs XWAITL and XDREQL (AS) are sampled by the rising edge of XPCLK These inputs have a set-up and hold requirement of 0ns and 10ns respectively

Appendices

Data Organisation - Big and Little Endian

The Jaguar system is intended to be usable in either a little-endian, e.g. Intel 80x86, or big-endian, e.g. 680x0, environment. The difference between these two systems is to do with the way in which bytes of a larger operand are stored in memory. There is potential for considerable confusion here, so this section attempts to explain the differences.

When storing a long-word in memory, a big-endian processor considers that the most significant byte is stored at byte address 0, while a little-endian processor considers that the most significant byte is stored at byte address 3. When both 32-bit processors are fitted with 32-bit memory this is not an issue for the memory interface, as the concept of byte address has no meaning; where it does become a problem is when the data path width is narrower than the operand width.

This document adopts the big-endian convention and Motorola operand ordering convention. Little-endian and Intel operand conventions could equally well have been applied.

IO Bus Interface

The IO Bus Interface is a 16-bit interface. Therefore, 32-bit data such as addresses will be presented differently between the little-endian and big-endian systems. What happens, in effect, is that the sense of A1 is inverted between the two systems. A big-endian system will see the high word of long-word at the low address, a little-endian system will see the high word at the high address.

Co-Processor Bus Interface

As the co-processor bus interface is 64-bits wide, there is no problem regarding big and little endian systems, although graphics processor programmers should always use byte, word, or long-word transfers as appropriate to the operand size to avoid having to be aware of whether the CPU is big or little endian.

Pixel Organisation

One side effect of the big or little endian philosophies is with regard to the organisation of pixels within a phrase.

In the little-endian system, the left-most pixel is always the least significant. In a phrase of data the left-most pixel includes bit 0. In byte address terms, this is in byte 0.

0 7	8 15		48 55	56 63

left right

In the big-endian system, the left-most pixel is always the most significant. The left-most pixel therefore always includes bit 63. In byte address terms this is stored in byte 0.

63	56	55	48		15	8	7	0

left right

Consider an eight-bit per pixel mode:

- in pixel mode, the left-most pixel in both systems is at byte address 0.
- in phrase mode, the little-endian left hand pixel is on bits 0-7, the big-endian left hand pixel is on bits 56-63.

(these modes refer to Blitter operation, which is described elsewhere)

This difference therefore affects operations that involve addressing pixels within a phrase when transferring a whole phrase at once (Blitter phrase mode).

Differences between Tom & Jerry and the Jaguar prototype

This is a summary of the major differences between the Jaguar prototype silicon and the Tom & Jerry devices, as an aid for programmers converting from one system to the other. Anyone writing system initialisation code should re-write it from scratch, referring to this manual.

Attempt to fix all published bugs.

All bugs of level 1 upwards in the Jaguar Rev 3 & 4 documentation should be fixed. Most of these fixes should be transparent to the user, where the fix has involved modifications to the programmer's view, the change is given below.

All the GPU and Blitter programming restrictions given in the previous bugs list are lifted. The extra NOPs required and illegal instruction sequences given in that bugs list can all now be disregarded.

Of the level 0 bugs, 4 is covered by the new blitter bus priority, 7 is unchanged, and 9 is covered by a new mechanism, see below.

Modify addresses for new 16 Mbyte address map.

The GPU/Blitter section follows the new ROMHI address map. This means internal registers start at F02000. This will now be consistent between all processors and bus masters. The system should always be run with ROMHI set to achieve this consistency.

Modify bus prioritization

The blitter, the DSP and the GPU can all now run at two priority levels. The previous blitter could only run at a lower priority than the object processor, but now by setting the BUSHI control bit in the command register it will request the bus at a higher priority than the Object Processor. This is particularly useful when doing something that involves lots of short blits, such as polygon rendering.

Better detection of Blitter completion

Allow blitter completion to be polled for, and correct timing of blitter interrupt generation for true completion. There is an IDLE bit in the blitter status register which flags true completion, i.e. the last bus transfer is completely terminated. The interrupt occurs as the IDLE bit is set.

Division of 16.16 bit numbers

Allow division on 16.16 bit numbers with 16.16 bit result - an extra mode bit selects this, DIV_OFFSET in the divide unit control register.

Different DMA request / acknowledge mechanism

If the GPU or the DSP is to be used as a software DMA controller, the mechanism now is that the DMA request is a suitable interrupt pin, and the DMA acknowledge, if a dedicated acknowledge line is required by the hardware, should be a dedicated GPIO line.

Blitter CRY pixel adding - 1

The ADDDSEL bit now allows blitter to add source and destination data, and writes this sum into RAM. See the discussion of the blitter command register.

Blitter CRY pixel adding - 2

The SRCSHADE bit uses the IINC register to modify source data, and may be used in conjunction with GOURZ for modifying the intensity of texture mapped surfaces (e.g. Tiger cube). See the discussion of the blitter command register.

PACK and UNPACK GPU Instructions

These allow simple pixel averaging to be performed. Unpack separates a 16-bit pixel so that the intensity is in bits 0-7, and the two colour fields are in bits 13-16 and 22-25. Other bits are set to zero. Pack reverses this, setting the top 16 bits to zero. This allows 2, 4, 8, 16 or 32 pixels to be averaged by unpacking them, adding them together, shifting right appropriately, and then packing them. See the Pack and Unpack section.

Blitter PITCH improvements

Version 1 allows blitter pitch values of 1, 2, 4 and 8 phrases. This misses out the extremely useful pitch of 3, and therefore 8 phrase pitch has been dropped, and the code for 8 will now give a pitch of 3.

Blitter Gouraud Z and Intensity ports

The blitter provides 8 new 32-bit write ports, which allow the 16.16 bit intensity and Z values computed at the start of a Gouraud strip to be written as single values, rather than splitting and combining them for the intensity and Z integer and fraction phrases. These ports just provide an alternative mapping of the existing source data (intensity integers), pattern data (intensity fractions), source Z1 (Z integers) and source Z2 (Z fraction) registers. Writing to the new intensity ports does not modify the colour byte fields.

SAT24 GPU instruction

A new GPU instruction saturates to a 24-bit unsigned integer This is useful for calculated intensities, which are often 8 point 16 bit numbers

Cartridge protection mechanism

When the GPU comes out of reset a LOCK bit is set which prevents the CPU doing anything to the GPU except setting the GO bit to execute from ROM address FF0008 When some software requirements in the ROM are met then the lock is cleared and the system starts up While the lock is set the blitter and object processor are disabled, and the GPU is invisible to the CPU The lock mechanism will not be documented in any more detail than this

Better Object Addressing

The object processor now contains a data field which allows object anywhere in the 16 Mbyte address space This obviates the need for ODP The extra bit is at the expense of the link address, which means object lists are now restricted to a 4 Mbyte area

Better Clock Generation and Control

Clock generation is now a function of Jerry, and a video clock divider has been added to the VMODE register in TOM

TOM and JERRY Bugs List

This document lists the known bugs in the TOM and JERRY devices This is revision code 2 silicon.

Level

3 This bug completely prevents some part of the ASIC from operating Some functionality cannot be demonstrated, and further bugs could be obscured

2 This bug can be fixed to some extent by a software or hardware work-around The functionality may still be impaired but is demonstrable

1 This bug can be fixed by a simple software or hardware work-around with no significant loss of functionality or performance

TOM Bugs

1 Unscaled 16-bit Object Fetch Speed

Level 1 *software*

Description Unscaled object data fetches for 16-bit objects occur every three ticks They should occur every two

Work-around None necessary. Will have a small impact on system performance.

2 Scoreboard Failure on Indexed Addressing Mode Stores

Note - This bug applies to both Tom & Jerry.

Level	1 *software*
Description	The data of indexed store instructions is not subject to any score-board protection. This means that the data written may not reflect what the programmer intended if the data is the result of a long latency instruction, that is divide or external load. It does not apply to the results of internal loads, moves or ALU operations as these are written back in time for the store. This bug applies only to store instructions 49, 50, 60 and 61. The full score-board protection still applies to the addressing registers.
Work-around	When storing data using these modes another instruction dependent on the store data should be placed ahead of the store, e.g.

```
div   r0,r3            ; long latency instruction
or    r3,r3            ; protection instruction
store r3,(r14+6)       ; write out quotient
```

This situation should be uncommon.

3 Transparency with HILO set

Note - this bug is only present on a few early test samples of Tom (Tom version 1), and is not present on any current production devices (Tom version 2). If you experience this bug it may be possible to have the Tom in your system upgraded from version 1 to version 2.

Level	2 *software*
Description	At the lowest level pixels are dealt with in pairs. If the pixels are displayed from high bits to low bits (HILO) then the transparency attributes are swapped. This is a new bug caused by the fix to previous HILO and scaling problems.
Work-around	The work around is to insist on transparent pixels appearing in pairs on even pixel boundaries. If this is unattractive, then double the width of existing images (thereby causing transparent pixels to occur in pairs) and display with a horizontal scaling factor of 0.5.

4 Horizontal Period register

Level	0 *hardware*
Description	With a 32MHz clock rate this register is only just long enough to achieve the 64us video line length.

5 Clipping inefficiency

Level	1 *software*
Description	If the number of displayed pixels is a lot less than 720 (the width of the line buffers) then wide objects, especially expanded 1 bit

objects, are not clipped on the right hand of the screen but at the end of the line buffer. This reduces the performance when these objects are on display.

6 Asynchronous BG can crash the bus arbitration state machine

Level	1	*hardware*
Description	BG is sampled in more than one place, and therefore if it occurs close to a clock edge the bus arbitration state machine can crash.	
Work-around	BG may need external synchronisation.	

7 No provision for auto vector (no VPA pin)

Level	0	*hardware*
Description	All interrupts are acknowledged with vector $40	

8 FC[0..2] should be ignored when Jerry owns the bus

Level	0	*hardware*
Description	These signals have to be tied off with resistors, as otherwise Tom can assume Jerry bus master cycles are the wrong type.	

9 SRCSHADE only works if GOURZ is set

Level	1	*software*
Description	For the SRCSHADE function to operate correctly the GOURZ flag must also be set. No Z data needs to be calculated or written, but the data paths are not set up correctly unless GOURZ is set.	
Work-around	Always set GOURZ when SRCSHADE is set.	

10 Blitter Pointer Read Registers are at the wrong address

Level	1	*software*
Description	The blitter pointer registers, which are written at addresses F02220C and F02230, appear for read at F02204 and F0222C. This error was also present on version 1 silicon.	
Work-around	Read them at the incorrect addresses.	

11 Blitter Y Add Control Bits

Level	2	*software*
Description	The Y add control bits in the A1 and A2 address generators in the blitter are not differentiated between properly. If the A1 Y add control bit is set it will affect both address generators. However, if the Y sign bits are set in either address generator, the corresponding add control	

bit still has to be set for the number to be negative. This error was also present on version 1 silicon.

Work-around: Either do not use this function, or use it on both address generators.

12 JERRY Bus Grant Pulses

Level: 1 *hardware*

Description: Tom can grant the bus to Jerry for one-tick wide periods. This can cause Jerry to incorrectly store write data for an impending write cycle, and consequently the write that Jerry goes on to perform when it gets the bus properly has the wrong data. These one-tick bus grants should not occur.

13 Scoreboard failure on successive writes

Note - This bug applies to both Tom & Jerry.

Level: 0 *software*

Description: If two instructions write to the same register with no read references to it in between, and the second of the two completes before the first, then the register can be left holding the result of the first, which is now what the programmer would expect. This is because there is no scoreboard protection against an instruction writing to a register that is currently flagged as invalid.

It was never envisaged that this situation would actually occur, but some programmers have managed to contrive circumstances under which it can occur, particularly when debug code is inserted, e.g.

```
load  (r3),r2          ; get data value
moveq 3,r2             ; over-write with bebug value
```

This combination can have the appearance of the MOVEQ instruction not being executed, as the load data is written into r2 after the quick immediate data.

Work-around: Put an instruction dependent on the data between the two, e.g. an `or r2,r2` between the two instructions above.

14 Single-stepping with MOVEI as first instruction

Note - This bug applies to both Tom & Jerry.

Level: 1 *software*

Description: The bug occurs when a MOVEI instruction is executed which empties the pre-fetch queue while in single-step mode. This can only occur (I think) at the start of an external program. The pre-fetch queue is two long-words, and a state machine attempts to keep it full, so it always contains three or four instruction words if no instructions are being used, i.e. after a short period in single-step stopped mode. On the first MOVEI, the instruction is executed as soon as the first long-word is fetched, which empties the pre-fetch queue again — so causing the problem.

Work-around: Do not start programs which are to be single-stepped in external memory with a MOVEI instruction.

15 Executing JUMP or JR from external memory

Note - This bug applies to both Tom & Jerry.

Level	3 *software*
Description	If either a JUMP or JR instruction is executed from external memory it is possible for this to align with the memory interface in such a way that the pre-fetch queue ends up holding invalid data. This means that these instructions can not be safely executed out of external memory.
Work-around	Do not place programs that contain JUMP or JR in external memory. This rules out almost all programs.

16 High Long Word Register

Level	2 *software*
Description	There is no scoreboard protection for the GPU high long word register. This causes various problems. If doing successive STOREP instructions, there is no way of telling when one has completed so that the high data can be loaded for the next one, this has the effect that successive STOREP instructions are really only useful when they write the same data. All external loads will modify this register, so that an interrupt which performs external loads will corrupt the high data from an underlying LOADP instruction, and there is no way for the interrupt service routine to preserve this data.

17 ADDDSEL or SRCSHADE with Z-buffering

Level	2 *software*
Description	If Z-buffer operation is enabled at the same time as the ADDDSEL or SRCSHADE bits are set, then the data is some-times corrupted. The only work-round known is to break the blit operation into two blits, one to do the SRCSHADE or ADDDSEL into an off-screen buffer, and then the second to perform the Z-buffer operation onto the screen. The failure mechanism is believed to be a pipe-line alignment issue, so that the data adders are being used for both Z calculation and data calculation at the same time, as these operations occur at different pipe-line stages. This failur mechanism has not been confirmed.

18 Blitter A1 Clipping Problem

Level	1 *software*
Description	If the A1 window clip register X value does not lie on a phrase boundary, then clipping occurs in the phrase on the right hand side of the clip window regardless of the state of the A1_CLIP bit.
Work-around	Zero the A1 window clip register when the A1_CLIP function is not in use.

19 Source shifts in 2-bits per pixel

Level	1 *software*
Description	If blitting 2-bit-per-pixel data and the source and destination are not aligned, the operation will fail at some alignments.

Work-around	Shift in one bit-per-pixel mode, or avoid doing this altogether

20 A1 clipping and DSTA2

Level	1 *software*
Description	If DSTA2 is set, the A1 clipping window will still affect the destination pointer in much the same way as bug 18
Work-around	When DSTA2 is being used with A1 clipping (DISO_A1), ensure that the clip window is a whole number of phrases wide

21 32 bit DSP is treated as 16 bit by data path

Level	1 *hardware - THIS DOES* **NOT** *AFFECT THE JAGUAR CONSOLE*
Description	In a 32-bit system, the Dsp is still treated as 16 bit by the byte control logic This means data is not presented properly for reads and writes
Work-around	?

22 RMW Object last pixel corruption

Level	1 *software*
Description	It is possible for the last column of pixels of an RMW object to be corrupted if it is followed by another pixel object This will be on the right unless the REFLECT bit is set This is due to a pipe-lining problem with some control signals
Work-around	Any of the following - ensure the last pixels of the source data are all transparent, i e pad the object data - make sure the next object in the list will not appear on the same line of the display - place an unused padding branch instruction after the object

23 GO bit may only be cleared locally

Level	1 *software*
Description	The GPU and DSP GO bits may only be cleared by the local processor The effect is intermittent and only pronounced if interrupts are running
Work-around	Require the local processor to clear the GO bits If necessary use a semaphore for another processor to signal it to stop

24 No Bus Master may operate at higher priority than the Object Proc.

Level	2 *software*
Description	Neither the DMAEN bit in the GPU nor the BUSHI bit in the Blitter may be set, because this can disturb the object processor If a higher priority bus master gets the bus between the second and third phrase of an object header then the line buffer address can be corrupted This will disturb the screen, usually appearing as horizontal black stripes

25 Consecutive divides fail

Note - This bug applies to both Tom & Jerry.

Level 1 *software*

Description There is a bug in the divider: if it tries to do two consecutive divides without there being one clock cycle of idle between them, then the result of the second divide will be wrong. This is because the internal divide length counter is not reset properly unless the divider has at least one clock cycle of inactivity between divides.

This will **only** occur when two divide instructions are separated by less than 16 clock cycles **and the second divide has the quotient of the first as one register operand**, and there is no score-board dependency on the quotient of the first one prior to the second.

Work-round Either make sure that more than 16 clock cycles occur between divide instructions, or make sure that an instruction which is dependant on the quotient of the first divide occurs before another divide. For example

This code should be like this

```
div     r0,r1     div     r0,r1
moveq   #3,r5     moveq   #3,r5
div     r5,r1     or      r1,r1
                  div     r5,r1
```

This code should be like this

```
div     r0,r1     div     r0,r1
div     r1,r2     or      r1,r1
                  div     r1,r2
```

26 Z Comparators fail in pixel mode without BKGWREN

Level 1 *software*

Description If the blitter is operating in pixel mode with the Z comparators enabled, then the comparator will not inhibit writes correctly. This can result in some pixels being written incrrectly, or in pixels not being written that should be.

Work-round The BKGWREN mode still works correctly. If DSTEN and BKGWREN are both set, then un-modified destination data is written back correctly. This will always solve the problem, although there is a speed penalty.

27 The Z registers can be shifted if SRCEN is set

Level 1 *software*

Description If doing a DSTWRZ blit with SRCEN set, but not SRCENZ, and a set of Z values are written into the Z registers, then the Z values get shifted as if they were read souce Z values. This has the effect of shifting the integer parts, and shifting up some of the fractional parts into the integer fields.

The Z shifting should clearly not be enabled unless SRCENZ is set, but in fact it is enabled if SRCEN is set.

Work-round	This only occurs if the source and destination are not phrase aligned One work-round, therefore, would be to pre-align the source data (in phrase mode)

28 A1 Clipping can clip one write too soon

Level	1 *software*
Description	When using the A1 clip mechanism, and A1 is the destination pointer, then the window clipping can occur one write (phrase or pixel) too soon, in a fairly random manner so that the right hand edge of the blit can sometime flicker The problem does not arise if DSTA2 is set
Work-around	There are three possibilities 1) If the clipped area is the screen, use a blitter window at lease one phrase or pixel wider than the displayed object(depending on the blitter mode desired), and set the clip window to one phrase or pixel wider The error will then occur outside the displayed area, but clipping further to the "right" will still work 2) Set DSTA2 if you can 3) Enable Z buffer writes This may move the problem from the pixel value to the Z value, which may still cause problems

29 A1 Clipping can fail to clip properly

Level	1 *software*
Description	This is very similar to bug 28 The reverse effect can occur, that is a pixel can fail to be clipped if the next one is not clipped If the increment values are large, the pixel that fails to be clipped my be a long way off from the target area, and may corrupt other data in RAM
Work-around	As for bug 28 None of these work-rounds are very satisfactory, unfortunately

Lies and Damned Lies

It is alleged that

- A jump then an indexed store/load causes a crash when interrupts are enable (Paul Foster & ATD)
- You can't jump to an MMULT directly, it needs a NOP first (Paul Foster)
- There have to be two instructions between MMULTs (Paul Foster)
- An MMULT must not follow a jump (obvious)
- We've found that you can't put the IMASK clear in the delay slot of the jump out of the interrupt, because the instruction that was interrupted may not get the correct register bank (TWI - Brian McKee)

JERRY Bugs

Note - check the TOM list, as bugs that apply to both the GPU and the DSP are listed there

1 RESETIL is a CMOS input

Level 1 *hardware*

Description The RESETIL input is a normal CMOS threshold input, it should be a Schmitt trigger input to avoid noise on power up.

Work-around Add an external Schmitt trigger buffer (e.g. two LS14 stages).

2 DSP slave reads only work at IOSPEED = 3

Level 1 *software*

Description Reads from DSP space in Jerry by another processor (slave reads) only work if the IOSPEED in MEMCON1 is set to 3, which gives 6 clock cycles for IO transfers (note that all manuals up to Rev 5 incorrectly document this as 2 clock cycles).

This is because the read data is only valid for one tick from the DSP itself, and it is not latched.

Work-around Always read from Jerry DSP space (F1A000 - F1FFFF) with IOSPEED set to 3. If slower peripherals are present in the system, IOSPEED will have to be dynamically altered.

3 Jerry can see previous DBGL

Level 1 *hardware*

Description If Jerry asserts DSP bus request one cycle after a previous bus request it is possible for it to see the end of the previous bus grant for one cycle, and this can mean that Jerry writes occur with the wrong data. The work-around is to ensure that Jerry is off the bus before performing a write, either by leaving a long period of bus inactivity, which is usually greater than the maximum possible period of object processor bus ownership, or to perform a load and perform an operation on the loaded data so that the score-board unit can ensure the load has completed.

4 Jerry generates long transfer size bits wrongly

Level 1 *hardware*

Description If Jerry does a long transfer in a 32 bit system, the size bits are 11 where they should be 00. Either only perform word transfers, or fix this externally.

5 Jerry does not look at the MASKA bits

Level 1 *hardware*

Description Jerry does not have the MASKA inputs from Tom, so long transfers from a 32 bit CPU are not recognised properly. The 32 bit CPU should perform word transfers.

6 DSP matrix multiplies only work in low 4K of RAM

Level 1 *software*

Description The DSP matrix address register can only point at locations in the first 4K of RAM. Only address line 2-11 are programmable, the rest of the matrix address is hard-wired as F1BXXX.